The Dissertation Committee for Ali Hussein Ibrahim
certifies that this is the approved version of the following dissertation:

Practical Transparent Persistence

Committee:

William R. Cook, Supervisor

Don Batory

Antony Hosking

Daniel Miranker

Keshav Pingali

Practical Transparent Persistence

by

Ali Hussein Ibrahim, B.S., M.Eng

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2009

Dedicated to my parents, Dr. Hussein Ibrahim and Dr. Nihal Nounou.

Acknowledgments

Many people have contributed to the work in this thesis. I owe a lot of thanks to my advisor, William Cook, who mentored and supported me while I worked on my research. I would also like to thank Ben Wiedermann with whom I collaborated with on several projects related to my thesis. Finally, I would like to thank my thesis committee, Don Batory, Daniel Miranker, Keshav Pingali, and Tony Hosking, for their advice and guidance.

On the personal side, I would have been lost without my wife, Sommayah, who supported and encouraged me during the entire Ph.D process. And of course, none of this would have been possible without my parents.

Practical Transparent Persistence

Publication No. _____

Ali Hussein Ibrahim, Ph.D.
The University of Texas at Austin, 2009

Supervisor: William R. Cook

Many enterprise applications persist data beyond their lifetimes, usually in a database management system. Orthogonal persistence provides a clean programming model for communicating with databases. A program using orthogonal persistence operates over persistent and non-persistent data uniformly. However, a straightforward implementation of orthogonal persistence results in a large number of small queries each of which incurs a large overhead when accessing a remote database. In addition, the program cannot take advantage of a database's query optimizations for large and complex queries. Instead, most programs compose smaller queries into a single large query explicitly and send the query to the database through a command-level interface. These explicit queries compromise the modularity of programs because they do not compose well and they contain information about the program's future data access patterns. Consequently, programs with explicit queries are harder to maintain and reason about. In this thesis, we first define transparent persistence, a relaxation of orthogonal persistence. We show how

transparent persistence in current tools can be made more practical by developing AutoFetch. The key idea in AutoFetch is to dynamically observe a program's data access patterns and use that information to reduce the number of queries. While AutoFetch is constrained by existing Java technology and tools, Remote Batch Invocation (RBI) adds the *batch* statement to the Java language. The batch statement is a general purpose mechanism for optimizing distributed communication using batching. RBI-DB specializes the ideas in RBI for databases. Both of these ideas help bridge the performance gap between orthogonally persistent systems and traditional database interfaces.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Much of the world's critical information infrastructure is built by combining general-purpose programming languages with database management systems (DBMS). These *enterprise applications* manage a constant stream of transactions and user interactions in support of our government, business, and personal interests. They combine a mixture of online transaction processing (OLTP) and online analytic processing (OLAP). Building, managing, and maintaining these applications is a primary focus of a large portion of the software developers active today, spanning commercial software companies, global consulting firms, and corporate information technology departments. Often, programmers build these systems using object-oriented languages such as Java, C#, C++, Python, and Ruby for general-purpose computation and relational databases for persistence. Relational databases excel at managing concurrent access to data, efficient querying, reliability, and flexibility.

Unfortunately, building systems using object-oriented programming languages and relational databases is fraught with difficulty because of the differences between their semantic foundations. These differences are known informally as *impedance mismatch* [103]: imperative programs versus declara-

tive queries, compiler optimization versus query optimization, algorithms and data structures versus relations and indexes, threads versus transactions, null pointers versus nulls for missing data, and different approaches to modularity and information hiding. Databases are often shared between several applications requiring careful thought about how to model data and share common functionality. Because relational databases and object-oriented programs are often structured as distributed systems, developers must make difficult architectural decisions about how to organize system functionality. Distributed execution also requires efficient structuring and management of specialized communication patterns.

1.1 Call-Level Interfaces

Bridging the impedance mismatch gap is left to the programmer when using *call-level interfaces* [151]. A call-level interface is a data access library that allows the programmer to send commands to a database management system. Call-level interfaces are the dominant approach to persistent data access in object-oriented languages. ODBC [87] and JDBC [75] are call-level interfaces for relational databases in C++ and Java respectively. Call-level relational database interfaces are a pragmatic approach to persistence: they support *explicit queries*. Explicit queries are queries written in the native language of the persistent store which for relational databases is SQL. The result of executing SQL queries is a list of tuples of primitive values which can be processed by the client program.

To illustrate the usage of call-level interfaces we present a simple example. The UML class model in Figure 1.1 shows a simple model for persistent data. The model can be translated into a relational database schema as in Figure 1.2. There are three classes: Department, Employee, and Company each of which is stored in a table. Each entity has a `name` field which is a string. Employees have a `salary` field which is real number. An employee may have a supervisor employee which is represented as a foreign key in the employee table. Companies have an employee who is the CEO which is represented as a foreign key in the company table. Each department has a set of employees represented as a foreign key in the employee table. Similarly, each company has a set of departments represented as a foreign key in the department table. A foreign key in a table represents an association between that table and the referenced table. For a collection association, the foreign key is in the table for the collection elements.

The code in Figure 1.3 shows how a Java programmer can program using this data model using the JDBC call-level interface. The code adds a department to a company, prints the CEO's salary for that company, and finally prints the name of all employees whose salary is greater than the CEO's salary.

JDBC and other popular call-level interfaces have many shortcomings.

1. The programmer must manage the resources associated with communicating with the database such as closing a `ResultSet` after it is not

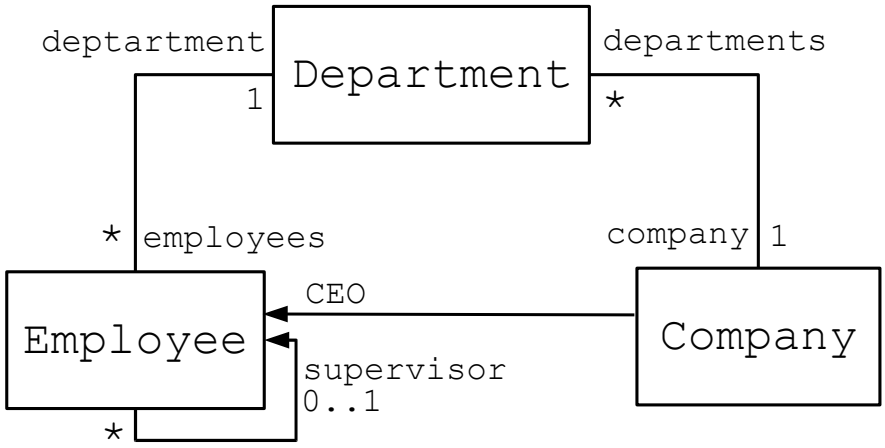

Figure 1.1: UML diagram for data model.

```
create table company (
  name varchar PRIMARY KEY,
  ceo varchar REFERENCES employee(name));

create table employee (
  name varchar PRIMARY KEY,
  salary double,
  dept varchar REFERENCES department(name),
  supervisor varchar REFERENCES employee(name));

create table department (
  name varchar PRIMARY KEY,
  company varchar REFERENCES company(name));
```

Figure 1.2: Relational database schema script in SQL for data model

```
Connection conn = ... ;
PreparedStatement ps = conn.prepareStatement(
 "insert into deparment (company_id, name, ...) " +
 "values ('Acme Widget', 'FooWidgets', ...)");
ps.executeUpdate();
ps = conn.prepareStatement(
 "select salary from employee e inner join company c" +
 " on c.ceo = e.id where c.name = 'Acme Widget'");
ResultSet rs = ps.execute();
rs.next();
double salary = rs.getDouble("salary");
rs.close();
System.out.println(salary);
ps = conn.prepareStatement("select e.name as empName, "
 + " d.name as deptName from employee e"
 + " left outer join department d on e.dept = d.name"
 + " left outer join company c on d.company = c.name"
 + " where c.name = 'Acme Widget' and e.salary > ?");
ps.setDouble(1, salary);
rs = ps.execute();
String empName = null;
while (rs.next()) {
  String empName = rs.getString("empName");
  String deptName = rs.getString("deptName");
  System.out.println("Employee " + empName
    + " in department " + deptName
    + " salary is higher than CEO.");
}
rs.close();
conn.close();
```

Figure 1.3: A code sample using JDBC. Error handling and transaction handling are omitted for simplicity.

needed.

2. Queries and results are not statically typed; the queries and the surrounding program may fail at runtime.

3. String manipulation of queries may result in security problems such as SQL injection.

4. Queries themselves are written in a different language from the program and cannot benefit from abstraction mechanisms in the language such as procedural abstraction.

5. The code is non-modular in the sense that splitting the code over several methods may necessitate complex string manipulation to produce a complete query.

The first three points are a result of poor language support for explicit queries and there have been successful attempts to address them. Gould et al. [71] implemented a type checker for explicit SQL queries expressed as strings. SafeQuery [46] expresses explicit queries as specialized Java methods. LINQ [50] incorporates explicit queries as a first-class construct into C#. We will focus on last two drawbacks which are a result of programmers having to split the program into explicit queries and program code.

1.2 Orthogonal Persistence

Orthogonal persistence was introduced in the 1980's as a way to avoid the impedance mismatch between programming languages and databases. Orthogonal persistence states that persistence behavior is independent of (orthogonal to) all other aspects of a system. The persistence may be implemented with any database management system. Atkinson and Buneman [12] defined orthogonal persistence as the following properties:

1. All types may be persisted equally.
2. All values are treated uniformly with respect to persistence. A value should be interpreted the same whether it is persistent or not. Programs manipulating persistent or non-persistent data should look the same.
3. Values are persisted with their type.

One consequence of these principles is that explicit queries should only be allowed in an orthogonally persistent program if they can be executed for in-memory objects as well as persistent objects. As a result, most orthogonally persistent architectures do not support explicit queries.

In an object-oriented language, persisted values are objects. The types of values are classes which define the behavior of objects. Thus orthogonal persistence in an object-oriented language should preserve the definition of a class along with its objects. Atkinson et al. [9, 13] further refined orthogonal persistence for object-oriented languages to include two additional principles:

1. The mechanism for persisting objects should be orthogonal to the universe of discourse of the programming language.

2. An object is persistent if and only if it is transitively reachable from a persistent root object.

A key characteristic of orthogonal persistence is that objects are loaded when needed. Loading a persistent object into memory is also called *faulting*. Using a reference in an object is called *traversing* the reference, or *navigating* between objects – such that the target object is loaded if it is persistent and not in memory. We use the term *navigational query* to refer to queries that are generated implicitly as a result of navigation.

As with objects in-memory, an object stored persistently can be deleted once it cannot be referenced. There is no explicit delete operation as in traditional interfaces. Concurrency is managed using the same mechanisms used for in-memory objects. Figure 1.4 shows how the example in Figure 1.3 could be written in PJama [13] an orthogonal persistence architecture for Java. In this context, the UML class model in Figure 1.1 defines the employee, department, and company classes and their associations.

The code in Figure 1.4 loads a root persistent object, a `Company` object, on line 3. On line 4, a `Department` object is created and added to the company. That department object becomes persistent because of transitive persistence. On line 5, the program navigates to the CEO employee object. This results in a navigational query against the underlying persistence store. On line 7, the

```
try {
  PJavaStore pjs = PJavaStore.getStore();
  Company c = (Company)pjs.getPRoot("Acme Widget");
  c.addDepartment(new Department("FooWidgets"));
  Employee ceo = c.getCEO();
  double salary = ceo.getSalary()
  System.out.println(salary);
  for (Department d : c.getDepartments()) {
    for (Employee emp : d.getEmployees()) {
      if (emp.getSalary() > salary) {
        System.out.println("Employee " + emp.getName()
          + " in department " + d.getName()
          + " salary is higher than CEO.");
      }
    }
  }
} catch (PJSException e) { ... }
```

Figure 1.4: A code sample for PJama.

program prints the salary of the CEO. Finally on lines 8–15, the program prints the name and department of all employees in the company who have a higher salary than the CEO. This code sample illustrates some of the strengths of orthogonal persistence. The code is written in the normal object-oriented style. The code is modular, the code may be easily broken into multiple methods each of which may perform navigational queries and update persistent state. The `println` method can treat the CEO salary as an in-memory object when in fact it is a persisted value. It also does not require the programmer to learn SQL; instead the programmer uses the familiar language constructs of Java such as loops and conditionals. The impedance mismatch is largely avoided by forcing the type system and the access style of the persistent store to support the programming language type system and access style.

Orthogonal persistence has been implemented, to a degree, in a variety of programming languages [113, 13, 99, 106]. It has found some success especially in computer aided design and embedded applications. One difference between programming with call-level interfaces and orthogonal persistence is that both data and behavior are persistent. Because behavior is persistent, data evolution and multi-language support is more difficult. Another difficulty is the requirement that all objects can be persisted. For example, it is difficult to persist objects that depend on the external environment such as threads and graphical user interface objects, because their state is tied to dynamic properties of an external system. Loading these objects requires recreating a consistent external state along with the recreating the internal state of the

object. Atkinson [11] highlighted this difficulty in the development of PJama.

1.3 Transparent Persistence

Orthogonally persistent architectures typically rely on object databases to serve as the persistent store. Orthogonal persistence has been somewhat successful in the field of computer-aided design where the navigational access style is natural and has good performance. As far as the authors know, no orthogonally persistent architecture uses relational databases as the persistent store. We surmise this is because of the expensive cost of single object faults and the difficulty in storing class or type information in available relational databases. However, relational databases are the focus of this thesis for several reasons. One is that they are entrenched, many businesses are committed to large relational databases shared among many applications. It is difficult to maintain several databases simultaneously while maintaining data coherency. Another reason is that relational databases are especially suited to enterprise application demands such as searching and aggregating large amounts of data.

Transparent persistence generalizes orthogonal persistence to preserve the programming style of orthogonal persistence, but remove some of the restrictions that make using relational databases difficult. All orthogonally persistent architectures are also considered to support transparent persistence. The following four points describe how transparent persistence relaxes the principles of orthogonal persistence:

1. Every type does not have to be persistence capable. A transparently persistent architecture may support the persistence of only a few predetermined types or types which have a certain structure.

2. The behavior of an object does not have to be persisted, i.e. only the fields of an object have to be persisted. Of course, when an object is loaded, some behavior must be associated with it, but this behavior may be different from the behavior associated with the object when it was persisted. For example, the most recent class definition for the object may be used.

3. The lifetimes of persisted objects may not depend solely on reachability or transitive persistence. An object may be added or removed from the persisted store with explicit commands. Of course, a persistent object may not be removed if it is referenced by another persistent object.

4. A transparently persistent architecture may allow the program to distinguish between persistent and non-persistent data through reflection or certain libraries.

Transparent persistence still preserves the programmatic style of orthogonal persistence. The persistent data is accessed by navigating from a persistent root datum. For object-oriented languages, transparent persistence preserves the representation of persistent data as an object graph which can be navigated through object fields or slots. We argue that this representation

is the core value of orthogonal persistence and will serve as our ideal programmatic model for the rest of this thesis. The code in Figure 1.4 is also representative of the programmatic model of transparent persistence.

Unfortunately, transparent persistence is not practical for many enterprise applications. The key issue is performance. Assuming no there is no prefetching or caching, the transparently persistent code performs $3+m$ faults, where m is the number of departments in the company. This does not include the creation of the new department. There are a number of assumptions in this calculation. We assume primitive field accesses such as name or salary fields do not cause faults. We also assume that accessing a collection results in a fault for that collection and all of its contained objects. Assuming that object faults are expensive, it is troubling that the number of faults grows linearly with the number of departments.

Call-level interfaces minimize round-trips by shipping code in the form of explicit queries to the relational database. Intermediate data used to compute the query may never be shipped to the client program, and data that is needed by client program is transferred in bulk as the result of the query. These explicit queries are also heavily optimized by the database query compiler. This pattern is a common optimization in distributed systems referred to as remote evaluation. On the other hand, transparent persistence offers no mechanism to take advantage of a relational database's general querying capability or to reduce the number of round-trips. David Maier [103] highlights this problem as one that must addressed by a successful programming model:

> "Whatever the database programming model, it must allow complex, data-intensive operations to be picked out of programs for execution by the storage manager, rather than forcing a record-at-a-time interface."

1.4 Contributions

We have claimed that transparent persistence provides a nicer programming model than call-level interfaces. However, programmers use call-level interfaces because they achieve higher performance by taking advantage of a relational database's querying capabilities. A natural direction then is to try to improve the performance of transparently persistent programs. In this thesis we describe two different approaches in this direction.

AutoFetch improves performance and modularity for existing hybrid object persistent architectures which combine explicit queries and navigational data access. Building on Hibernate [79], AutoFetch optimizes explicit and navigational queries by adding prefetch directives to prevent future object faults. In doing so, it allows for more modular programs. The programmer is also freed to use a more transparently persistent programming style.

On the other end of the spectrum, Remote Batch Invocation (RBI) solves the more general problem of optimizing programs that make remote procedure calls. We develop an new programming language construct `batch` which allows the programmer to demarcate a block of client code as describing a remote operation which can be batched in one round-trip to the server. This

batch block can combine remote and local code with some data-flow restrictions. A simple functional language is used to describe remote operations and is executed on the server using a form of remote evaluation. For database access, we compile this functional language to SQL queries to leverage the query optimizations in a relational database.

1.5 Performance

Since our goal is to provide a programmer with a language and abstractions which achieve good performance, it is important to identify the key considerations. Our core assumption is that in client-server architectures, the cost of executing a query, which involves a round-trip to a database, typically dominates other performance measures. This is because the latency cost of communicating with the database is significantly greater than the cost of processing the query or producing results [18, 107]. Other factors, the number of joins, subqueries, or columns returned from a query, are insignificant compared to latency. The relative impact of latency on system performance is likely to increase, given that improvements in latency lag improvements in bandwidth and processing power [116]. As a result, number of queries will increasingly dominate all other concerns. In effect, overall response time is directly related to the number of queries executed in a task. Therefore, we focus on minimizing the number of queries executed by a program, while validating that approach in our evaluation.

Chapter 2

Autofetch

AutoFetch improves the modularity and performance of *existing* programs written with *object relational mapping* (ORM) libraries. These ORM libraries allow the programmer to use both navigational and explicit queries to access persistent data. Programmers using ORM tools can combine both styles of data access in their programs, but must be careful to avoid performance degradation due to navigational queries. Navigational queries can be reduced by using *prefetch* directives added to preceding explicit and navigational queries. However, programmer specified prefetch does not scale to multiple modules and imposes an additional programming burden on the programmer. AutoFetch is an extension to object relational mapping tools that automatically adds prefetch directives to queries.

2.1 Object Relational Mapping

AutoFetch extends Hibernate [79] an ORM tool. Other examples of ORM tools include EJB [108], JDO [123], and Toplink [60]. ORM tools allow the programmer to program against a persistent object model using relational databases. In this model, the relational database is viewed as a graph of

objects. The programmer must specify a mapping between the object model and the relational database model. There are a variety of mapping idioms [6]. For example, we will assume that each tuple defines a single object and that associations are defined using foreign keys. The object tuple contains all the primitive (strings are considered primitive in this context) fields of the object. Figures 1.1 and 1.2 show an example of a correspondence between classes and relations.

Mapping inheritance relationships is more complicated. Since AutoFetch operates at the level of object graphs, the mapping of inheritance hierachies is orthogonal to the work in this thesis. Ambler [6] has a good discussion of the various options and tradeoffs.

Once the mapping is in place, the ORM tool transparently translates between objects and tuples. Most ORM tools also allow explicit queries over the object model that are translated to SQL and executed by the relational database. While query languages can significantly increase performance, they reduce orthogonality because they are special operations that only apply to persistent data. The results of queries are translated into a sub-graph of the persistent object graph and the program is given a list of *root* objects from that sub-graph. If the program navigates beyond this sub-graph, the ORM tool executes navigational queries to load objects as needed. This feature allows the ORM tool to support transparent persistence. The following is an example of the Hibernate Query Language (HQL), an explicit ORM query language.

```
select distinct(p) from Person p left join fetch p. children
  where p.parent.firstName='John'
```

HQL combines features of the object query language (OQL) and SQL. It supports *path expressions* such as `p.parent.firstName` which denote the traversal of fields in the object model. An HQL query is compiled to SQL given a particular mapping of an object model to relational tables. In particular, path expressions which involve non-primitive fields (i.e. fields which themselves are not tuples) are translated to SQL joins between the tables of the source object's class and the destination object's class.

The `fetch` keyword indicates that related objects should be loaded along with the main result objects. In this query, the children for each person that matches the query criteria are prefetched. Consequently, the program will not incur additional navigational queries if it accesses the children association for the persons returned by the query. As expected, the `left outer join fetch` translates to a SQL left outer join. This strategy causes the data for the container object to be replicated when a collection association is fetched. For a nested collection, the root container is replicated once for each combination of subcollection and sub-subcollection items. Thus replication is multiplied with each level of subcollection. Independent fetch collections are especially expensive because they cause the result set to include the cross-product of independent collection hierarchy elements. If Hibernate used a different query strategy that allowed for multiple SQL queries to be executed, while correlating the results in the client, then this problem

could be eliminated. The results of fetch joins are not visible in the top-level query results; rather these results are used to populate the associated objects corresponding to the fetch.

Prefetch of related objects is especially important in addressing the $n + 1$ select problem in which a related object is accessed for each result of a query. Without prefetch, if there are n results for a query, then there will be $n + 1$ loads. Most JDO vendors extended the standard to allow prefetch to be specified at the class level. Hibernate, and now EJB 3.0, allow prefetch to be specified within each query using the `fetch` keyword. These related objects can be either single objects or collections of related objects, depending on whether the association is single- or multi-valued.

A program may lock objects or the results of a query. The ORM does not manage such locks, rather any program locking commands are translated to the corresponding database locking commands.

The ORM tool also manages object identity. Each object is guaranteed to be unique within a unit of work. In Hibernate, the unit of work is a *session*. Once a persistent object is loaded in memory, any future references to that object will resolve to a single reference within the same session. Hibernate enforces this property by storing each object keyed by its *identity* in a session cache. The identity of objects is defined in the mapping to relational database tuples and corresponds to the primary key in a relational table.

The Java code in Figure 2.1 shows one way the code in Figures 1.3

```
1  Company c = (Company)session.load("Acme Widget");
2  c.addDepartment(new Department("FooWidgets"));
3  Employee ceo = c.getCEO();
4  double salary  = ceo.getSalary()
5  System.out. println ( salary );
6  Query q = session.createQuery(" from Employee e "
7   + " left outer join fetch e.department"
8   + " where e.department.company.name = 'Acme Widget'"
9   + " and e.salary > :salary");
10 q.setDouble("salary", salary);
11 List<Employee> emps = (List<Employee>) q.list();
12 for (Employee emp : emps) {
13   printEmp(emp);
14 }
15  ...
16 void printEmp(Employee emp) {
17   System.out. println ("Employee " + empName
18     + " in department " + emp.getDepartment().getName();
19     + " salary is higher than CEO.");
20 }
```

Figure 2.1: Using Hibernate ORM library to access persistent data.

and 1.4 can be written in Hibernate. Lines 1–5 are similar to the transparent persistence style code in Figure 1.4. In lines 6–9, the program executes an explicit query. The query returns a list of Employee objects as the root objects and prefetches the department associations for those objects. A single explicit query is executed after line 5; the navigational queries on line 18 are avoided because of the prefetch directive in the query. Without the prefetch directive, the program would execute $n + 1$ queries where n is the number employees returned by the explicit query.

This program combines transparent persistence and call-level interfaces by leveraging both explicit queries and navigational data access. The explicit query allows the programmer to exploit the query optimizer in the relational database and the navigational access allows the programmer to have the `printEmployee` method be unaware of the persistence or non-persistence of the data.

2.2 Problem

While specifying prefetch manually in a query can significantly improve performance, correct prefetch specifications are difficult to write and maintain manually. The prefetch definitions on line 7 in the query in Figure 2.1 must correspond exactly to the code that uses the results of the query (lines 10 through 21).

It can be difficult to determine exactly what related objects should be prefetched. Doing so requires knowing all the operations that will be performed

on the results of a query. Modularity can interfere with this analysis. For example, the code in Figure 2.1 calls a `printEmp` method which can cause additional navigations from the employee object. It may not be possible to statically determine which related objects are needed. This can happen if class factories are used to create operation objects with unknown behavior, or if classes are loaded dynamically.

As a program evolves, the code that uses the results of a query may be changed to include additional navigations, or remove navigations. As a result, the query must be modified to prefetch the objects required by the modified program. This significantly complicates evolution and maintenance of the system. If a common query is reused in multiple contexts, it may need to be copied in order to specify different prefetch behaviors in each case.

Since the prefetch annotations only affect performance, it is difficult to test or validate that they are correct – the program will compute the same results either way, although performance may differ significantly.

In this thesis, we present and evaluate AutoFetch, which uses traversal profiling to automate prefetch in object persistence architectures. AutoFetch records which associations are traversed when operating on the results of a query. This information is aggregated to create a statistical profile of application behavior. The statistics are used to automatically prefetch objects in future queries.

In contrast, previous work focused on profiling application behavior in

the context of a single query. While this allowed systems such as PrefetchGuide [76] to prefetch objects on the initial execution of query, AutoFetch has several advantages. AutoFetch can prefetch arbitrary traversal patterns in addition to recursive and iterative patterns. AutoFetch can execute fewer queries once patterns across queries are detected. AutoFetch's disadvantage of not optimizing initial query executions can be eliminated by combining AutoFetch with previous work.

When applied to an unoptimized version of the Torpedo benchmark, AutoFetch performs as well as a hand-tuned version. For the OO7 benchmark, AutoFetch eliminates up to 99.9% of queries and improves performance by up to 85%. We also examined several case studies; a sample application for web applications, a blogging platform, and a promotional website. In all of these case studies, AutoFetch allows the programs to be simpler and perform as well as a less-modular, hand-optimized version.

Safe Query Objects are a type-safe alternative to string-based query interfaces [47]. Safe queries use methods in standard object-oriented languages to specify query criteria and sorting, so that a query is simply a class. Unlike string-based query languages, there is no natural place to specify prefetch in a Safe Query. Thus Safe Queries would benefit significantly from automatic prefetching.

2.3 Automating Prefetch

In this section we present AutoFetch, a solution to the problem of manual prefetch in object persistence architectures. Instead of the programmer manually specifying prefetches, AutoFetch adds prefetch specifications automatically. By profiling traversals on query results, AutoFetch determines the prefetches that can help reduce the number of navigational queries.

To formalize this approach, we define type and object graphs as an abstract representation of persistent data. A type graph represents the class model, or structure of the database. Object graphs represent data. A complete database is represented as an object graph. Queries are functions whose range is the set of subgraphs of the database object graph.

Traversals represent the graph of objects and associations that are actually used in processing each result of a query. These traversals are aggregated into *traversal profiles*, which maintain statistics on the likelihood of traversing specific associations. Queries are classified into *query classes* based on a heuristic that groups queries that are likely to have similar traversals.

For each query executed, AutoFetch computes a prefetch specification based on the traversal profile for its query class. The prefetch specification is incorporated into the query and executed by the underlying database.

2.3.1 Profiling Traversals

In this section we develop a model for profiling the traversals performed by an object-oriented application. The concept of *profiling* is well known [15, 65]; it involves collecting statistics about the behavior of a program. Profiling is typically used to track control flow in an application – to find hot spots or compute code coverage. In this paper, profiling is used to track data access patterns – to identify what subset of a database is needed to perform a given operation.

We develop a formal model for types, objects, queries, and traversals. The type and object models are derived from work on adaptive programming [97].

2.3.1.1 Type Graphs

A type graph is a directed graph $G_T = (T, A)$.

- T is a set of type names.
- F is a set of field names.
- A is a partial function $T \times F \xrightarrow{?} T \times \{single, collection\}$ representing a set of associations between types. Given types t and t' and field f, if $A(t, f) = (t', m)$ then there is an association from t to t' with the name f and cardinality m, where m indicates whether the association is a single- or multi-valued association.

Each edge in the type graph represents an association between a single source object and some number of target objects. Inheritance is not modeled in our type graph because it is orthogonal to prefetch. Bi-directional associations are modelled as two uni-directional associations. The formal representation of the data model in Figure 1.1 is shown below.

T = {Department, Employee, Company}
F = {employees, departments, CEO, supervisor}
A(Department, employees) $\mapsto$ (Employee, collection)
A(Company, departments) $\mapsto$ (Department, collection)
A(Company, CEO) $\mapsto$ (Employee, single)
A(Employee, supervisor) $\mapsto$ (Employee, single)

2.3.1.2 Object Graphs

Let O be the finite set of object names. An object graph is a directed graph $G_O = (O, E, G_T = (T, A), \mathit{Type})$. G_T is a type graph and *Type* is a unary function that maps objects to types. The following constraints must be satisfied in the object graph G_O:

- O represents a set of objects.
- $\mathit{Type} : O \rightarrow T$. The type of each object in the object graph must exist in the type graph.
- $E : O \times F \xrightarrow{?} \mathit{powerset}(O)$. The edges in the graph are a partial function from a source object and a field to a set of target objects.
- $\forall o, f$: $E(o, f) = S$

- $A(Type(o), f) = (T', m)$
- $\forall o' \in S$, $Type(o') = T'$.
- if $m = single$, then $|S| = 1$.

Each edge in the object graph corresponds to an edge in the type graph. Single associations have exactly one target object. Collection associations have zero or more target objects.

An example object graph is shown in Figure 2.2 which is based on the type graph in Figure 1.1. In this model, collection objects are not represented as separate objects. Instead, we will abstract collections as multi-valued edges represented by a dark ovals. This is because ORM tools normally fault collections and their elements together. There are some exceptions to this, for example, some ORM tools allow just the size of a collection to be faulted into memory, however, this is not widely implemented or used.

Each edge in an object graph corresponds to a possible navigational query. Therefore, null-valued single associations are not represented by edges in the object graph, because most ORM tools store the nullity of an association in the source object. This is natural when working with relational databases, because single-valued optional associations are represented by a foreign key in the source object. The foreign key column will be null for a null-valued association and this is loaded along with the object's other primitive fields. On the other hand, empty collection associations are represented as a multi-valued edge with no values. Again this choice is related to how most ORM

tools work. ORM tools query the database for empty collection associations, because the source object does not have any information on the cardinality of the collection.

2.3.1.3 Queries

A query consists of an *extent type*, a *criteria*, and a *prefetch specification*. The extent type is the type of the root objects that will be returned when the query is executed. The criteria are the conditions that an object of the extent type satisfies to be returned as a root object. The prefetch specification specifies which objects to prefetch for each root object. When a query is executed the ORM tool returns to the user a subgraph of the database object graph. This subgraph consists of a list of *root* objects and a set of *prefetch* objects. The root and prefetch objects are connected by all the edges between them in the full persistent object graph.

Our approach to prefetching is independent of a particular query language, however, the query language must support an object-oriented view of persistent data and prefetch specifications.

A navigational query is a query which is executed by the ORM tool to fault objects into memory and are not present in the program source as query strings. The criteria for navigational queries specifies a single object using its identity. Navigational queries may have prefetch specifications which come from annotations on the data model. For example, the programmer might specify that all navigational queries for employees should prefetch their

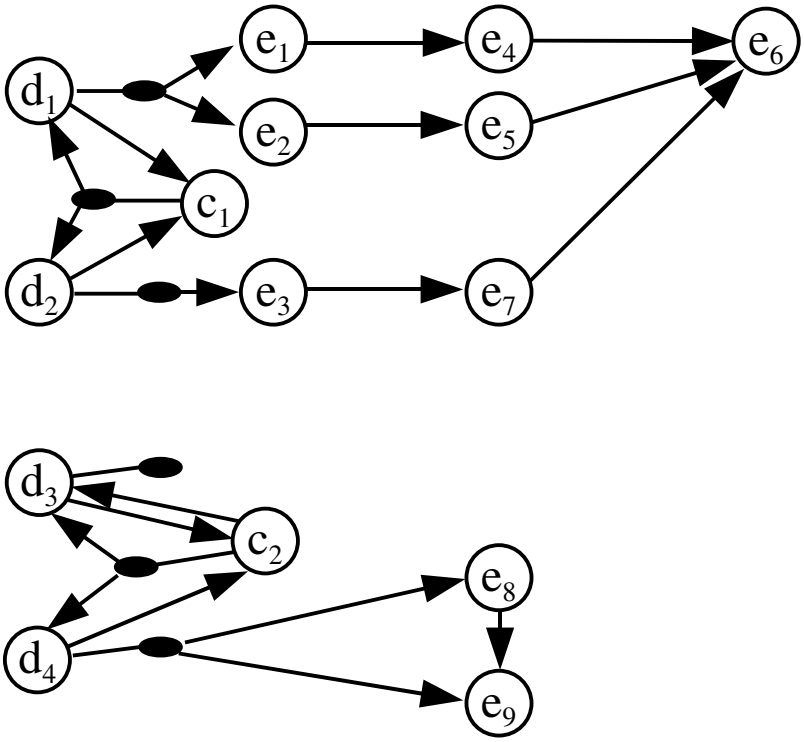

Figure 2.2: An example of an object graph based on the type graph in Figure 1.1. Collection associations contain an oval in the middle of the edge.

department.

Queries are executed by the program to return their results. However, queries are first-class values, because they can be dynamically constructed or passed or returned from procedures. A single program point could execute different queries, depending on the program flow.

2.3.1.4 Traversals

A traversal captures how the program navigates the object graph that the query returns as a set of trees. A node in the object graph is traversed if the program faults it into memory. An edge in the object graph is traversed if the program, without prefetch directives, executes a navigational query for that edge. A program may traverse all the objects and associations in the result of the query, or it may traverse more or less. If the program traverses outside of the boundary of the query result object graph, then navigational queries are executed to fault objects into memory.

A traversal is represented as a forest where each tree's root is a root object in the result of a query and each tree is a sub-graph of the entire object graph. Let R denote a single tree from the traversal on the object graph $G_O = (O, E)$:

$$R = O \times B$$

$$B = (F \xrightarrow{?} powerset(R))$$

For all $(o, b) \in R$, if $b(f)$ is defined then $e(o, f)$ is also defined. This ensures that traversal graphs mirror object graphs. If the program navigates to the same object using different paths in the object graph, only the first path from the root of the traversal is included in R. Figure 2.3 shows a sample traversal on the object graph in Figure 2.2 for a query which returned 3 departments: d_1, d_2, d_3. Edges with dark ovals represent collection associations. In this example, the program traverses the employees association and company associations for each department reaching e_1, e_2, e_3 and c_1, c_2 respectively. For department d_1, there is no associated company which could mean either that the program did not access the company field or that the company field was null. For department d_3, the employees association is empty and is represented by an empty multi-valued edge. The program then traverses the supervisor relationship for employees e_1, e_2, e_3 to varying depths to get to objects e_4, e_5, e_6. In the original object graph, e_6 is the supervisor for both e_4 and e_5, however the traversal will only include one of these edges even if the program uses both paths to reach e_6.

If a program traverses a target object cached in-memory by the ORM tool, then that object is omitted from the traversal. In addition, any single edges that target that object or paths starting from that object are omitted. This follows from our principle that traversal edges represent navigational queries; faulting a cached object does not require a navigational query. This illustrates an interesting link between caching and query execution; AutoFetch is able to adapt to the caching mechanism of the application by

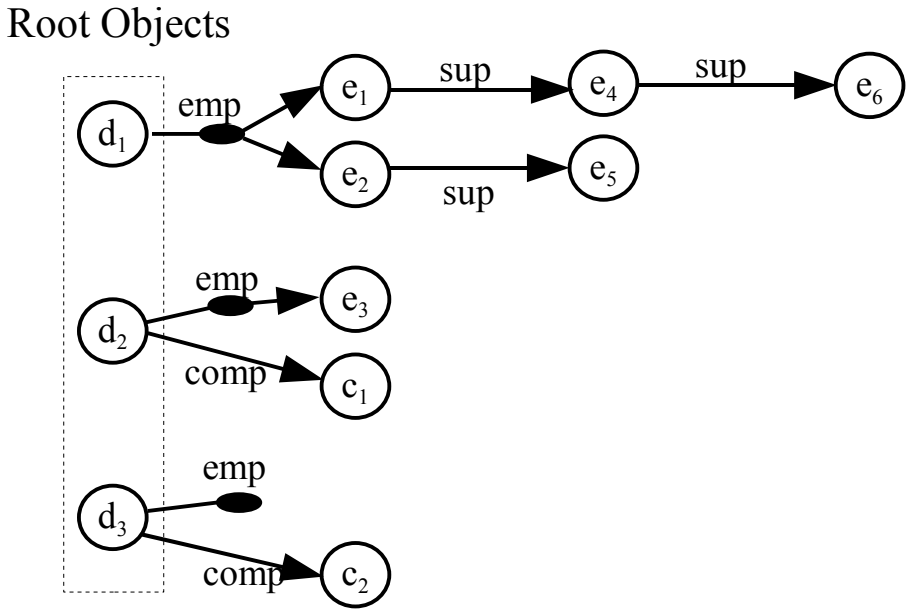

Figure 2.3: An example of a traversal on the object graph in Figure 2.2. Collection associations contain an oval in the middle of a multi-valued edge.

adjusting query prefetch to ignore navigating to objects that are likely in a cache.

Our representation of traversals abstracts some information about the program traversal such as the order in which objects were traversed. One future change we are considering is representing the fact the program traversed the first n objects in a collection. This would allows AutoFetch to issue more precise prefetch specifications in cases where the program only needs the first n objects in a collection association.

On the other hand, our representation of traversals is too low-level to capture semantic information about the program such as that a traversal is the result of a transitive closure on a graph of objects. Since the SQL standard does not support transitive closure (there are proprietary extensions), AutoFetch would not currently benefit from such information.

Since a program can only perform finite work in any period of time, traversals are also finite.

An important point is that a single query may be used in different contexts that generate different traversals. This will commonly happen if a library function runs a query to load a set of objects, but this library function is called from multiple transactions. Each transaction will have a different purpose and therefore may traverse different associations.

2.3.1.5 Traversal Profiles

A traversal profile represents an aggregation of the traversals. While traversals mirror the object graph, traversal profiles mirror the type graph. A traversal profile is a traversal of the type graph where each node can be visited more than once and edges are annotated with statistics. Let P represent a traversal profile for a type graph $G_T = (T, A)$:

$$P = T \times (F \rightarrow \mathbb{N} \times \mathbb{N} \times P)$$

such that for all $(t, \{(f, \mathit{used}, \mathit{potential}, p)\}) \in P$

1. $A(t, f)$ is defined. Edges in the traversal profile must correspond to edges in the type graph.

2. *used* $\leq$ *potential*. The *used* statistic must be less than or equal to the *potential* statistic.

Each edge in the tree contains statistics on the navigational queries. The *potential* statistic measures the number of times the source node of an edge was in memory. The *used* measures the number of times a navigational query would have been executed if there were no prefetch directives. By definition, *used* must be less *potential*, because a program can only cause a navigational query to occur for an edge in the object graph if it has the source object in memory. The *potential* statistic equals the *used* statistic of the preceding edge if the preceding edge represents a single valued association. For collection associations it represents the sum of the sizes of collections represented by the *used* statistic of the preceding edge. Figure 2.4 shows a traversal profile updated from an empty traversal profile and the traversal in Figure 2.3. The traversal profile statistics are given above each type as (used/potential).

A traversal can be incorporated into a traversal profile in which the types of the root objects are all the same type and that type is the root type of the traversal profile. The traversal, a forest of object trees R, is combined with a traversal profile using a function *combine*: $(R \times P \rightarrow P)$. Given a traversal and a traversal profile, *combine* produces an updated traversal profile. The basic idea is to overlay the traversal with the traversal profile

Algorithm 1 $combine(r : \{(o, AO)\}, (t, AP))$

if $|r| = 0$ **then**
  return (t, AP)
end if
for all $(t, f, t_{target}, card) \in G_T$ **do**
  $usedIncr = 0$
  for all $(o, AO) \in r$ **do**
    if $AO(f)$ is defined **then**
      $usedIncr = usedIncr + 1$
    end if
  end for
  $newR = \bigcup_{(o,AO)\in r} AO(f)$
  if $AP(f)$ is defined and $AP(f) = (used, potential, p)$ **then**
    $w(f) = (used + usedIncr, potential + |r|, combine(newR, p)$
  else
    $w(f) = (usedIncr, |r|, combine(newR, (t_target, \{\})$
  end if
end for
return (t, w)

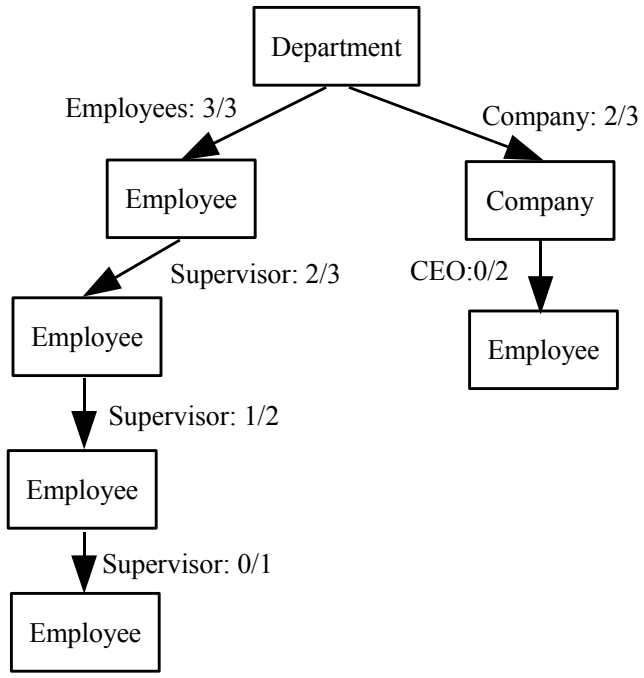

Figure 2.4: Traversal profile for a query class after traversal in Figure 2.3. Statistics are represented as (used/potential).

by mapping objects to their types. Figure 2.5 shows how the the traversal in Figure 2.3 can be overlayed with traversal profile in Figure 2.3. Once the two graphs are overlayed, we increment both the *used* and *potential* statistics on each matching edge. We also increment the *potential* statistic on each edge in the traversal profile that is not in the traversal, but its ancestor edge is in the traversal.

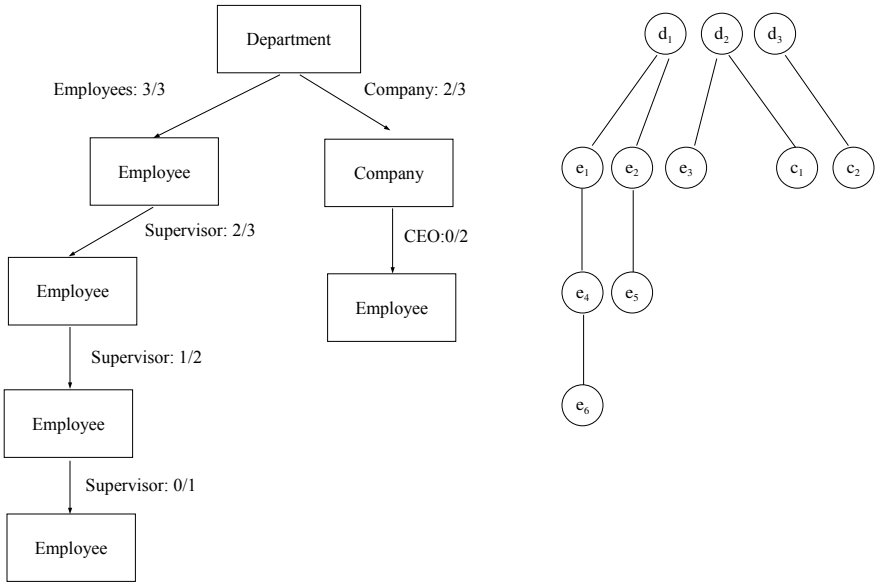

Figure 2.5: Overlaying traversal on traversal profile

Algorithm 1 shows pseudo-code which performs this overlay in a single pass over the traversal. The time complexity of this algorithm is linear in the size of the traversal. Traversals are assumed to be finite, so the algorithm is guaranteed to terminate.

2.3.2 Query Classification

Query classification determines a set of queries that share a traversal profile. The aim of query classification is to group queries which are likely to have similar traversals. A simple classification of queries is to group all

queries that have the same query string. There are several reasons why this is not effective.

First, a given query may be used to load data for several different operations. Since the operations are different, the traversals for these operations may be different as well. This situation typically arises when query execution is centralized in library functions that are called from many parts of a program. Classifying based only on the criteria will not distinguish between these different uses of a query, so that very different traversals may be classified as belonging to the same class. This may lead to poor prediction of prefetch. The classification in this case is too coarse.

A second problem is that query criteria are often constructed dynamically. If each set of criteria is classified as a separate query, then commonality between operations may not be identified. At the limit, every query may be different, leading to a failure to gather sufficient data to predict prefetch.

Queries may also be classified by the program state when the query is executed. This is motivated by the observation that traversals are determined by the control flow of the program after query execution. Program state includes the current code line, variable values, library bindings, etc. Classifying queries based on the entire program state is infeasible as the program state may be very large and will likely be different for every query. However, a set of salient features of the program state can be reasonable both in memory and computation. Computation refers to cost of computing the program state features when a query is invoked.

The line number where a query is executed is a simple feature of the program state to calculate and has a small constant memory size, however, it does not capture enough of the program state to accurately determine the traversal of the query results. Specifically the problem is that line number where the query is executed does not provide enough information on how the results of the query will be used outside of the invoking method.

The natural extension to the using the line number where the query is executed is using the entire call stack when the query is executed. Our hypothesis is that the call stack gives more information about the future control flow, because it is highly likely that the control flow will return through the methods in the call stack. The call stack as the salient program state feature is easy to compute and bounded in size. In the programs we have considered, we have found that the call stack classifies queries at an appropriate granularity for prefetch.

Unfortunately, a call stack with line numbers will classify 2 queries with different extent types together if the 2 queries occur on the same line. To address this, AutoFetch uses the pair of the query extent and the call stack when the query is executed to classify queries. This might conflate two different queries that have the same extent, but it preserves AutoFetch's ability to prefetch for many dynamic queries. Optimally, the call stack would contain information on the exact program statement being executed at each frame. Previously [84], we used the query string as part of the query class which does distinguish between dynamic queries. Which approach is best depends

on the program and is not relevant if the program does not put multiple query statements on a single program line.

2.3.3 Predicting Traversals

Given that an operation typically traverses a similar collection of objects, it is possible to predict future traversals based on the profiling of past traversals. The predicted traversal provides the basis to compute the prefetch specification. The goal of the prefetch specification is to minimize the time it will take to perform the traversal. A program will be most efficient if each traversal is equal to the query result object graph, because in this case only one round-trip to the database will be required and the program will not load any more information from the database than is needed. The heuristic used in AutoFetch is to prefetch any node in the traversal profile for which the probability of traversal is above a certain threshold.

Before each query execution, AutoFetch finds the traversal profile associated with the query class. If no traversal profile is found, a new traversal profile is created and no prefetches are added to the query. Otherwise, the existing traversal profile is used to compute the prefetch specification.

First, the traversal profile is trimmed such that the remaining tree only contains the associations that will be loaded with high probability (above a set threshold) given that the root node of the traversal profile has been loaded. For each node n and its parent node $p(n)$ in the traversal profile, the probability that the association between n and $p(n)$ will be traversed given that p(n) is

in memory can be estimated as *used/potential* where *used* and *potential* are the statistics defined on the edge between n and $p(n)$. Using the rules of conditional probability, the probability that the association is navigated given that the root node is loaded is:

$$f(n) = (used/potential) * f(p(n))$$

The base case is that the $f(root)$ in the traversal profile is 1. A depth first traversal can calculate this probability for each node without recomputing any values. This calculation ensures that traversal profile nodes are prefetched only if their parent node is prefetched, because for all n, $f(n) \leq f(p(n))$.

Second, if there is more than one disjoint collection path in the remaining tree, an arbitrary collection path is chosen and other collection paths are removed. Collection paths are paths from the root node to a leaf node in the tree that contain at least 1 collection association. This is to avoid creating a query which joins multiple independent many-valued associations.

The prefetch specification is a set of prefetch directives. Each prefetch directive corresponds to a unique path in the remaining tree. For example, given the traversal profile in Figure 2.4 and the prefetch threshold of 0.5, the prefetch specification would be: (employees, employees.supervisor, company). The query is augmented with the calculated prefetch specification. Regardless of the prefetch specification, profiling the query results remains the same.

2.4 Implementation

We have written two implementations of AutoFetch. The first implementation, described in a earlier paper [84], extends Hibernate by changing the Hibernate source code directly and supports prefetching for navigational and HQL queries. The second implementation, AutoFetch OS, extends Hibernate through its plugin architecture. Because Hibernate does not expose the internal HQL query structure, AutoFetch OS supports prefetching for *criteria* queries instead of HQL queries. Criteria queries are a programmatic way of expressing explicit queries and can express most HQL queries. AutoFetch OS sacrifices some performance for modularity because it cannot piggyback on Hibernate's internal query processing. We will discuss and evaluate AutoFetch OS, because it has been released as an open source project and is more robust for use by other researchers.

The AutoFetch OS implementation can be divided into a traversal profile module which stores the traversal profiles and calculates prefetch directives and a Hibernate 3.2 plugin which enables the dynamic profiling and applies the prefetch directives to queries. The architecture of AutoFetch OS is shown in Figure 2.6.

2.4.1 Traversal Profile Module

The traversal profile module maintains a 1-1 mapping from query class to traversal profile. When the Hibernate extension asks for the prefetch specification for a query, the module computes the query class which is used to

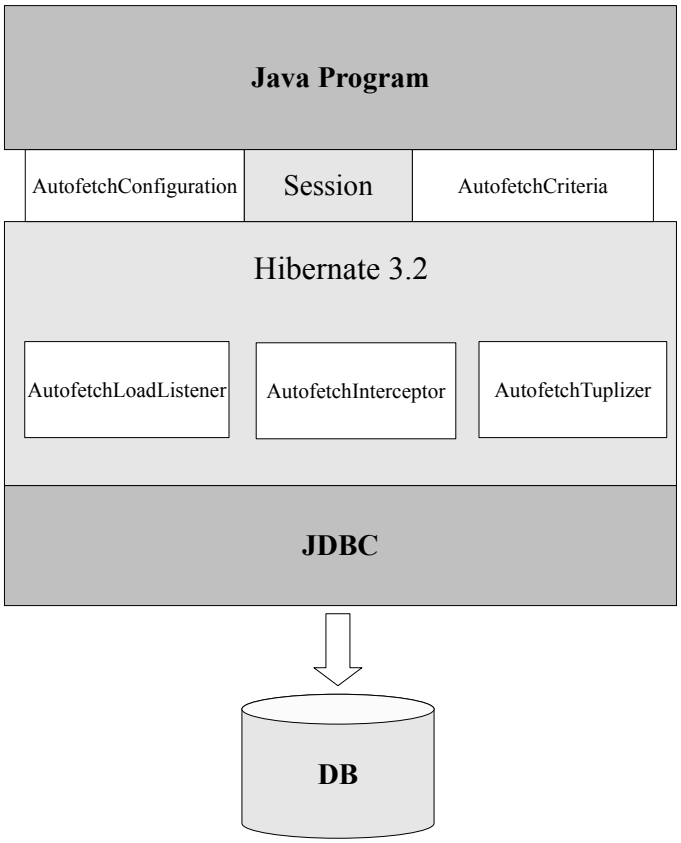

Figure 2.6: AutoFetch architecture.

lookup the traversal profile and compute the prefetch specification. The module computes the query class as the pair of the extent type and the current program stack trace and uses this as the key to lookup the traversal profile. The stack trace is filtered to remove any frames for method calls in the Hibernate or AutoFetch OS libraries, because we do not want queries to be differentiated by how the persistence architecture loads objects. To decrease the memory requirements for maintaining the set of query classes, each stack

trace contains a maximum number of frames. If a stack trace is larger than this limit, AutoFetch OS removes top-level frames until the stack trace is under the limit. We found that keeping 20 frames was sufficient for all of our benchmarks. Each frame is a string containing the name of a method and a line number. If a traversal profile does not exist for a query class, the module adds a mapping from that query class to an empty traversal profile.

If the traversal profile does exist for a query class, AutoFetch OS generates a prefetch specification for the query using the traversal prediction algorithm in Section 2.3.3. The prefetch specification is filtered such that no two collection associations are loaded if they are on different paths from the root of the traversal profile. This policy is to prevent an unconstrained join between the two collections which will cause a large increase in the size of the query result set. Another way to avoid this problem is to execute multiple queries.

After the query is executed, the Hibernate extension passes the results back to the traversal profile module. The results are then marked as the root of traversals for the appropriate query class so that traversals from those results are profiled correctly. As an optimization, AutoFetch OS employs *sampling*. Each result is profiled with a certain probability which is set to ten percent in our experiments. Sampling reduces the overhead of object access by reducing the number of traversals off of root objects that are monitored. In addition, we sample objects in a collection association such that only a subset of the collection association is profiled. At least one result is always profiled

in either case. Optimally, we would like to not proxy those root objects which are not sampled, however, the plugin architecture we are using does not allow the information flow needed to implement this.

2.4.2 Hibernate

The Hibernate AutoFetch OS extension is designed for Hibernate 3.2. Figure 2.6 shows the important plugins that we used to modify the behavior of queries in Hibernate. A couple of other plugins just make sure that we do not profile Hibernate's accesses of objects, e.g. when Hibernate checks if a modification has been made to a persistent object.

- The *AutofetchConfiguration* class decorates the Hibernate *Configuration* class to override the existing collection association types.
- The *AutofetchCriteria* class decorates the Hibernate *Criteria* class to implement profiling and prefetch for criteria queries.
- The *AutofetchInterceptor* uses the Hibernate interceptor extension point to maintain a reference to the traversal profile module and correctly unwrap the type of instrumented entities.
- The *AutofetchTuplizer* uses the Hibernate tuplizer extension point to instrument persistent objects to obtain traversal information.
- The *AutofetchLoadListener* uses the Hibernate listener extension point to implement profiling and prefetch for navigational queries.

- The *AutofetchPreLoadEventListener*, *AutofetchPostLoadEventListener*, *AutofetchAutoFlushEventListener*, *AutofetchDirtyCheckEventListener*, and *AutofetchFlushEventListener* classes make sure that object profiling is turned off when Hibernate is accessing objects.

The code in Figure 2.7 illustrates how a Criteria query is modified to include prefetches and the SQL generated by Hibernate for different prefetch directives. Using AutoFetch OS, the programmer does not specify the prefetch directives manually, but they are automatically added by the system.

Every persistent object is instrumented with a proxy. The proxy enables profiling by intercepting all method calls to an object. The proxy has two states: profiling and non-profiling. The proxy is in the profiling state unless the Hibernate library could manipulate the object in which case it is temporarily in the non-profiling state. While profiling, any method call other than the identifier getter causes the proxy to to mark the object as accessed and increment the *used* statistic in the appropriate traversal profile nodes. The set of traversal profile nodes associated with a proxy reflect the different traversal paths to an object. The set of traversal profile nodes does not contain two nodes from the same traversal profile; only the first path to an object is profiled for each query class. Thus, no traversal is explicitly computed, instead the proxy updates traversal profiles directly. This frees the implementation from determining when a traversal has been completed. Once a proxy detects an object access it updates the traversal profiles associated with

its sub-objects. Hibernate proxies (lazy loaded objects) are instrumented by modifying the Hibernate proxy class. Plain Java objects are instrumented using a dynamically generated proxy. Collections are instrumented by modifying the Hibernate collection classes.

AutoFetch OS does not support all of Hibernate's features. For example, AutoFetch OS does not support prefetching or profiling for data models which contain weak entities or composite identifiers. Support for these features was omitted for simplicity. As mentioned earlier, AutoFetch OS does not support prefetch for HQL queries, because Hibernate does not provide public access to the structure of HQL queries. While we could have implemented our own HQL query parser, we decided that prefetching criteria queries was sufficient since they can express most HQL queries. In our evaluation, criteria queries were able to express all the queries except for a single query expressing a non-natural join; i.e. a join not represented by an association.

2.5 Evaluation

We evaluated AutoFetch using several benchmarks and case studies. For the benchmarks, the programs are fixed and the performance is measured with and without AutoFetch. For the case studies, we also discuss how the program's design might be affected by AutoFetch.

Benchmarks:

Original query

Criteria:

```
Criteria crit = sess.createCriteria(Department.class);
crit.add(Restrictions.eq("name", "foo"));
```

SQL:

```
select * from Department as d where d.name = 'foo'
```

Query with a single prefetch

Criteria:

```
Criteria crit = sess.createCriteria(Department.class);
crit.add(Restrictions.eq("name", "foo"));
crit.setFetchMode("employees", FetchMode.JOIN);
```

SQL:

```
select * from Department as d
left outer join Employee as e on e.deptId = d.id
where d.name = 'foo'
```

Figure 2.7: Augmenting queries with prefetch specifications.

- The Torpedo benchmark measures the number of queries that an ORM tool executes in a simple auction application.

- The OO7 benchmark examines the performance of object-oriented persistence mechanisms for an idealized CAD (computer assisted design) application.

Case Studies:

- The Struts resume case study shows how AutoFetch interacts with an example application written by Matt Raible to illustrate different Java web technologies.

- The roller blog case study shows how AutoFetch interacts with a popular Java blogging web application.

- The Ebean website case study shows how an independent implementation of AutoFetch was used in a live website.

All the benchmarks and local case studies are run using a Postgresql 8.2 relational database engine. The database resides on an Intel®Pentium®4 2.4 Ghz machine with 1010 Mb of RAM. Clients for timed benchmarks are run on a separate machine; an Intel®Core®2 Duo running at 2.33GHz with 2022 Mb of RAM. Both machines reside on the University of Texas at Austin computer science network. The ping program reports a latency of 200 to 500 micro-seconds for a message of 64 bytes between the two machines. The Java clients run on the Sun 1.6 client JVM with a maximum of 1296 Mb of memory. We use the AutoFetch OS implementation for every benchmark and case study except for the Struts resume case study which uses the initial implementation instead and the Ebean website which uses an independent implementation of AutoFetch. We set the maximum extent level and stack frame limit parameters to 12 and 20 respectively unless otherwise noted.

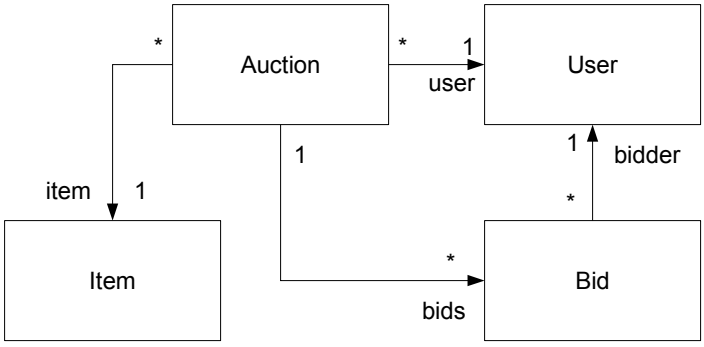

Figure 2.8: UML class diagram for Torpedo data model.

2.5.1 Torpedo Benchmark

The Torpedo benchmark [107] measures performance of Java ORM tools using an auction application. The auction application manages four types of persistent objects: Items, Bids, Users, and Auctions. Figure 2.8 shows the relationships between these types.

The benchmarks characterizes ORM tools using a single number: the number of queries executed for the benchmark use cases. There are eight different use cases each of which consists of one or more database transactions. Since the use cases only use the name of the user objects which is the primary key, they are not usually faulted from the database.

1. *List All Auctions* lists information about every auction including the bids and auction item. This use case has the potential to exhibit the

$n + 1$ query problem because it accesses the bids and item for the n auctions in the database.

2. *Find High Bid* finds the highest bids for a single auction and the bidders responsible for those high bids. In the data, this use case finds a single high bid.

3. *List Auction* lists the full information about an auction including the bids, bidders, auctioneer, and auction item.

4. *List Auction Twice Without Transaction* executes the List Auction client interaction twice in separate transactions. This use case is an opportunity for ORM tools to cache information across transactions.

5. *List Auction Twice With Transaction* executes the List Auction client interaction twice in a single transaction. This use case is an opportunity for ORM tools to cache information inside transactions.

6. *List Partial Auction* lists some information about an auction. Unlike the *List Auction* use case, it does not list information about the bids.

7. *Place a Bid* places a bid on an auction.

8. *Place Two Bids* places two bids by a single user on two different auctions. This use case is an opportunity for ORM tools to batch inserts.

The *List Auction*, *List Auction Twice with Transaction*, *List Auction Twice without Transaction*, *Place Bid*, and *Place Two Bids* use cases exhibit

the $n+1$ problem because of an interaction with caching. Normally, accessing the bids collection causes all of the bids to be loaded for an auction; however, if inter-transaction caching is available than Hibernate may only load the identifiers of the bids hoping to find the bids in the cache. If the cache lookup fails, then each bid is loaded separately.

Each use case is deterministic, the data requested is always the same. The database is small; it contains 3 auctions, 3 items, 14 users, and 20 bids. For each client interaction a SQL logger tool is used to count the number of queries executed including transactional commit statements. Thus the minimum number of queries for a client interaction is two. The benchmark produces a single performance number which represents the total number of queries executed running each client interaction one time.

We based our implementation of the benchmark on the reference implementation written by Bruce Martin [107]. The reference implementation allows different ORM tools to be swapped in by overriding the `Persistence` class. We created a parameterized version of this class for Hibernate which allowed us to test three different benchmark implementations.

1. A hand-optimized version which uses Hibernate without AutoFetch. We add prefetch directives to both the Hibernate configuration files and queries to minimize the overall number of queries. Our optimized version executes the same number of queries as the best published Hibernate result (the submission was for Hibernate 2.1.6). We will call this the

hand-optimized version.

2. An unoptimized version which uses Hibernate without AutoFetch. This version contains no prefetch directives in either the Hibernate configuration files or queries. Each association is lazily loaded. We will call this the unoptimized version.

3. An unoptimized version which uses Hibernate with AutoFetch. We will call this the AutoFetch version.

A fourth configuration is possible; an optimized version with AutoFetch enabled, but we do not expect AutoFetch to be useful in this case. We ran the Torpedo benchmark for each version twice in succession. The results are shown in Figure 2.9.

In the first iteration of the benchmark, the AutoFetch version executes as many queries as the unoptimized version, but subsequently it executes three times less queries matching the performance of the optimized version. Table 2.1 shows the breakdown by use case for the number of queries executed for the second iteration of each benchmark version.

Most of the additional queries are executed are due to the $n + 1$ select problem for the *Find All Auctions*, *List Auction Twice with Transaction*, *List Auction Twice without Transaction*, *Place Bid*, and *Place two Bids*. The *Place two Bids* use case executes less queries than the *Place Bid* use case because it adds bids to different auctions with less existing bids. The AutoFetch

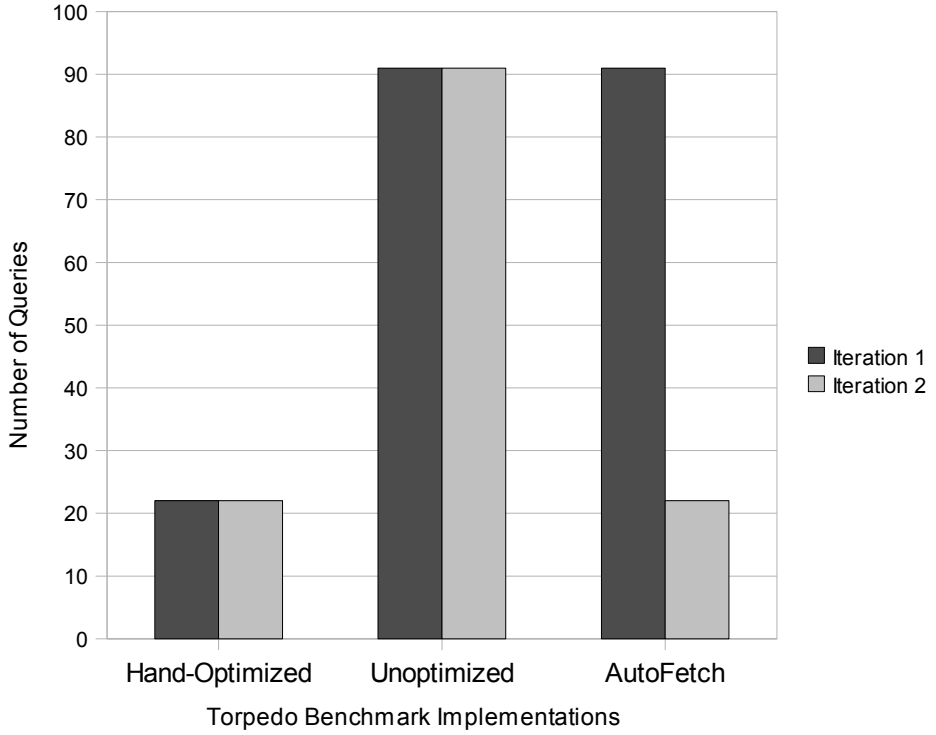

Figure 2.9: Torpedo benchmark results. The y-axis represents the number of queries executed. Maximum extent level is 12.

Use Case	Unoptimized	Hand-Optimized	AutoFetch
Find All Auctions	8	2	2
Find High Bids	2	2	2
List Auction	16	2	2
List Auction Twice wo Tx	17	3	3
List Auction Twice with Tx	16	2	2
List Partial Auction	2	2	2
Place Bid	18	3	3
Place Two Bids	12	6	6

Table 2.1: Comparison of the number of queries for different versions of Torpedo benchmark on second iteration.

version matches the hand-optimized version's performance on the second iteration once it has seen each use case once before. This is not surprising since the traversal patterns in the Torpedo benchmark are deterministic and shallow. This is encouraging since we believe that this is the most common type of traversal pattern in enterprise applications and the one most suited for AutoFetch.

By manual inspection of the logged queries, we observe that the AutoFetch version's queries are the same as the hand-optimized queries. This is important, because a trivial way to optimize the number of queries is to load the entire database into memory which is not feasible for more realistic databases.

Just using the line number where the query was executed as the query class would not have been sufficient to match the performance of the hand-optimized benchmark implementation. For example, the `findAuction` method is used to load both detailed and summary information about an auction.

The detailed auction information includes traversing several associations for an auction such as the auction bids. The summary auction information only includes fields of the auction object such as the auction id or date. These different access patterns require different prefetches even though they use the same backend function to load the auction.

2.5.2 OO7 Benchmark

The OO7 benchmark [34] was designed to measure the performance of object oriented database management systems (OODBMS). It evolved from the Hypermodel and OO1 benchmarks and is designed to test all aspects of an object database. Initially, Carey and Dewitt used the OO7 benchmark to compare different commercial OODBMS, however, the research community has extended its use to persistent languages and different programming paradigms. It is by far the most popular benchmark for research into persistent object systems. However, its deep traversals and flat queries are not well-suited to AutoFetch as we will see..

The benchmark consists of a series of traversals, queries, and structural modifications performed on a synthetic parts database. Three sizes of the database are specified: small, medium, and large. Each database consists of modules which contain two sub-structures. The assembly hierarchy is a complete trinary tree with seven levels. The parts graph is a set of composite parts each of which contains a distinct graph of atomic parts with one root part. The root atomic part is part of a connected graph of atomic parts with

Table 2.2: Comparison without AutoFetch OS and with AutoFetch OS. Maximum extent level is 12. Small OO7 benchmark. Metrics for each query/-traversal are the number SQL queries, time in milliseconds, and number of objects loaded.

Query	Iteration	No Autofetch			AutoFetch		
		queries	ms	entities	queries	ms	entities
Q1	1	11	9	10	11	18	10
	2	11	11	10	11	13	10
Q2	1	2	8	77	2	9	77
	2	2	9	77	2	9	77
Q3	1	2	28	1010	2	35	28
	2	2	33	1010	2	36	33
Q4	1	101	395	470	2	453	470
	2	101	416	457	2	422	416
Q5	1	2	17	238	2	19	238
	2	2	18	238	2	20	238
Q7	1	2	227	10000	2	255	10000
	2	2	208	10000	2	271	10000
Q8	1	2	24	103	2	25	103
	2	2	24	103	2	24	103
T1	1	20816	20817	41027	887	28898	41027
	2	20816	22598	41027	708	23433	41027
T6	1	1589	2630	2080	24	345	2080
	2	1589	2149	2080	1	301	2080
T8	1	2	13	1	2	12	1
	2	2	12	1	2	13	1
T9	1	2	12	10	2	12	10
	2	2	12	10	2	12	10

a constant degree of three, six, or nine. We use a fanout of three for our experiments. The number of atomic parts in the each graph varies from 20 to 200 based on the size of the database. Each leaf in the assembly hierarchy is linked to three unshared composite parts and three shared composite parts. Unshared parts may not be shared between different modules. There are some additional objects such as documents and manuals, but these do not figure prominently in our experiments.

Only a few of the OO7 operations involve object navigation, which can be optimized by AutoFetch. Traversal T1 is an almost complete traversal of the OO7 object graph of both the assembly and part hierarchies. It traverses the assembly hierarchy depth-first and then traverses the parts object graphs for the unshared composite parts. Traversal T6 traverses the entire assembly hierarchy, but only accesses the unshared composite parts and their root atomic part in the part hierarchy. Traversal T1 has a depth of about 29 while Traversal T6 has a depth of about 10. Neither the queries nor traversals T8 or T9 perform navigation; however, they are included to detect any performance penalties for traversal profiling. We do not include the results for Traversals 2a, 2b, and 2c because their results are similar to Traversal 1. They differ from Traversal 1 in that they perform updates during the traversal which does not interact with AutoFetch. We did not test the structural modifications because AutoFetch has no effect on inserts, updates, or deletes. Finally, we did not test Traversal CU because its performance is highly sensitive to the caching policy chosen.

We implemented a Java version of the OO7 benchmark based on code publicly available from the benchmark's authors. Explicit queries are represented using criteria queries except for Query 8. This query cannot be expressed using criteria queries; it includes a join not represented by an association. Therefore the results of Query 8 are not profiled, but they are proxied, so there is still some overhead from AutoFetch OS. When we tested our first implementation of AutoFetch [84] we mistakenly set the number of composite parts to 50 instead of 500 for the small database size. This causes the number of atomic parts to decrease ten-fold and greatly decreased the size of the results for Query 7 and Traversal 1 which in turn decreased the overhead of our profiling. In addition, it should be noted that our first AutoFetch implementation has lower overheads for inserting query prefetch and dynamic profiling because we changed the Hibernate code directly instead of relying on the public interfaces provided. Two variations of the benchmark were created, one that uses the AutoFetch OS plugin and one that does not. The variation without the AutoFetch OS plugin incurs no dynamic profiling overhead. Our measurements are for *hot* runs; specifically we run the benchmark five times before making our measurements. This was sufficient to reach a stable steady state for time measurements. For the AutoFetch variation, after the five warmup runs we clear the traversal profile information so that the sixth run is effectively the same as the first run in terms of prefetch. There is no inter-transaction caching, so the primary affect of the warmup runs is for the JVM to optimize methods and for the database static

parameters such as connections and query plans to be initialized and cached.

Table 2.2 summarizes the results of the OO7 benchmark for the small data size. Neither the queries nor traversals T8 or T9 navigate object associations; therefore, they do not require or benefit from prefetch. However, the results of these queries and traversals are dynamically profiled, so we expect to measure the overhead for AutoFetch OS. Query 7 shows the largest relative performance loss because it loads the most objects (10000) into memory, however, the overhead is still less than 25%. For all of the queries and traversals, AutoFetch OS does not load more objects than the version without AutoFetch OS.

Both traversals T1 and T6 show a large improvement in the number of queries even during the first iteration of the benchmark. This is because the AutoFetch OS is learning traversal profiles for each subpath in the traversals and using that information later in the traversal. Traversal T6 shows the expected large improvement in time taken; almost an order of magnitude improvement. Traversal T1 shows a slight degradation in time taken even though the number of queries is much less. This surprising result is due to several reasons:

1. T1 traverses the part hierarchies which are the largest part of the database and thus incur the most profiling overhead.

2. The queries that prefetch the object graph are inefficient because they fetch the same object along multiple paths.

3. Queries with deep prefetch are not efficiently executed by relational databases because the flat result format cannot efficiently represent the results.

The last point points to an inefficiency in the representation of SQL results. Some researchers [42] have explored how to encode these types of results more efficiently, which would greatly help in reducing the overhead of a collection association prefetches.

The behavior of Traversal T6 in Figure 2.10 shows that for smaller tree traversals, the number of queries and time are correlated. Figure 2.10 shows how the maximum depth of the traversal profile affects the number of queries executed and time for Traversal T6. The data points are for the sixth run of executing Traversal T6 for all the different maximum extent level values. Interestingly there is a local minima at a maximum depth of three. This is due to the structure of the assembly hierarchy. For prefetch depths of less than three, the traversal requires three sets of queries to be executed at various depths of the tree. For a prefetch depth of three, the traversal requires two sets of queries to be executed at levels 1 and 4 of the tree. For prefetch depths of five through seven, the traversal still requires two sets of queries executed at levels 1 and 5 – 7 of the tree, however, there are more nodes at levels 5 – 7 than at level 4. The cost of increasing the maximum depth of the traversal profile is an increase in the memory requirements to store traversal profiles and an increase in the complexity of queries. Because deep traversals such as

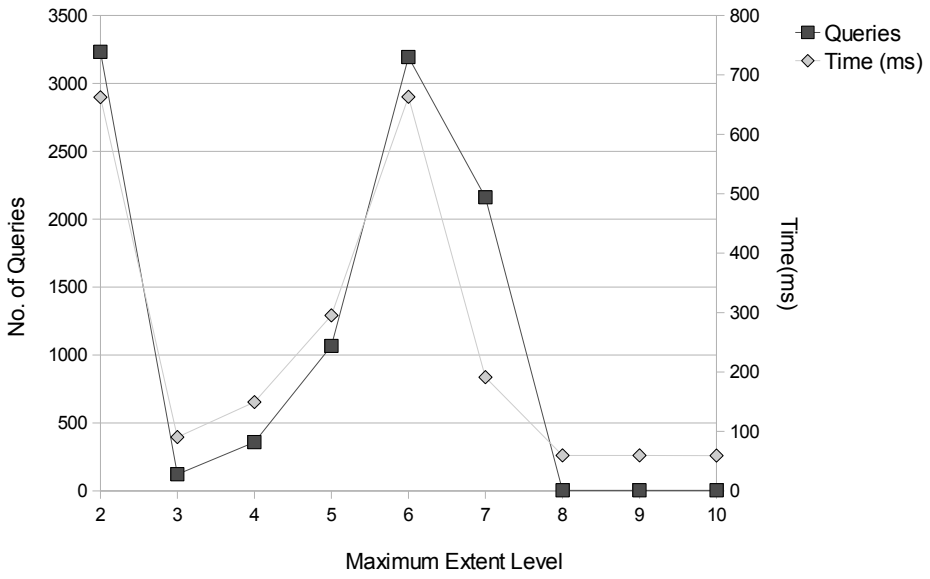

Figure 2.10: Varying maximum extent level from 5 to 15. Small OO7 database.

T1 and T6 in OO7 are relatively rare in enterprise business applications, we expect that prefetch levels of five or so to be sufficient for most cases.

2.5.3 Resume Application

In addition to the synthetic benchmarks, we applied AutoFetch to a resume application that uses the AppFuse framework [132]. AppFuse is a template for a model-view-controller (MVC) architecture that integrates many popular Java libraries and tools. AppFuse includes a flexible data layer which can be configured to use one of several persistence providers. Users of the framework define interfaces for data access objects (DAO) that are implemented using the persistence provider.

```
interface ResumeDAO {
  Resume getResume(Long resumeId);
  ...
}

class Resume {
  List getEducations() { ... }
  List getExperiences() { ... }
  List getReferences() { ... }
  ...
}
```

Figure 2.11: Struts resume code without any optimizations

Hibernate is used as the persistence provider in the sample resume application. The resume application data model is centered around a `Resume` class. A `Resume` contains basic resume data fields and associations to related objects, including education listings, work experiences, and references. The `ResumeDAO` class includes methods to load and store resumes. A simple implementation of the `ResumeDAO` and `Resume` classes is shown in Fig 2.11. The `ResumeDAO.getResume(Long)` method loads a resume without prefetching any of its associated objects. To load the work experience in a resume, a programmer first uses `ResumeDAO` to load the resume, and then calls the `getExperiences()` method to load the work experience.

Although this implementation is very natural, it is inefficient because the resume application has several pages that display exactly one kind of associated object; a page for work experience, a page for references, etc. For these pages, the application would execute 2 queries: one to load the resume

and another to load the associated objects. There are several alternative implementations:

1. Modify the `ResumeDAO.getResume(Long)` method to prefetch all associations.

2. Add `ResumeDAO` methods which load a resume with different prefetch directives.

3. Add `ResumeDAO` methods which directly load associated objects without loading the resume first.

The actual implementation uses the third approach. The first alternative always loads too much data and would be infeasible if the data model contained cycles. The other two alternatives are fragile and redundant. For example, if a new user interface page was added to the application that displayed two resume associations, then a new method would have to be added to the `ResumeDAO` class. The code is also redundant because we have to copy either the `ResumeDAO.getResume(Long)` method in the second alternative or the `Resume` getter methods in the third alternative. By incorporating AutoFetch, the simple code in Figure 2.11 should perform as well as the optimized code after some initial iterations.

We tested the code in Figure 2.11 with AutoFetch and found that indeed it was able to execute a single query for all the controller layer methods after the initial learning period. Our modified code has the advantage of being

smaller, because we eliminated redundant methods in `ResumeDAO` class. With AutoFetch, DAO methods are more general because the same method may be used with different traversal patterns. AutoFetch also increases the independence of the user interface or view layer from the business logic or controller layer, because changes in the traversal pattern of the user interface on domain objects do not require corresponding changes in the controller interface.

2.5.4 Java Roller Blog

Roller is a popular open source Java blog server. Its users include Sun Microsystems and the US government both of which use it to manage employee blogs. Roller allows users to manage their own blogs directly and allows outside visitors to post responses and comments. We modified Roller version 3.1 which is the last version to support Hibernate exclusively. Later the Roller codebase was changed to use JPA of which Hibernate is a single implementation.

Roller employs a lot of caching to get good performance. Caching is well-suited for this application, because most visitors to the site only view blog posts and do not update content. Unfortunately, caching obscures the effect of AutoFetch and makes repeated experiments difficult, so we disable all the different forms of caching that are present. Of course, this implies that AutoFetch's benefits may be limited in practice; only when the various cache entries are invalidated is the database queried.

We created three versions of the Roller application:

1. *Original*: The original code containing the static prefetch directives that the Roller programmers specified. AutoFetch OS is used, but prefetch is disabled.

2. *Lazy*: The original code with the static prefetch directives removed. AutoFetch OS is used, but prefetch is disabled.

3. AutoFetch: The original code with the static prefetch directives removed. AutoFetch OS is used and prefetch is enabled.

We then measure the number of queries needed to load a blog page. The statistics are collected by deploying the Roller application and accessing the web pages using the wget program (to avoid client side caching of pages). We collect data after hitting a page a couple of times to reduce time variability and allow AutoFetch OS to gather profiling data. Our sample blog server database contains four users, four blogs, twenty one blog posts, and thirty two comments.

Table 2.3 shows the number of queries, time, and number of objects faulted for each version of the Roller application while visiting the blog homepage. The blog homepage has 7 posts and 16 comments. The AutoFetch version executes almost the same number of queries as the original version and is more precise; it loads the minimum number of objects as specified by the lazy version. The reason why the original version loads more objects and executes more queries than needed is because the programmers specified a static prefetch policy which applies to all queries. No single prefetch policy is correct

Table 2.3: Comparison of Roller versions with respect to the number of queries, time, and the number of objects faulted into memory when loading the home page of a blog.

Version	No. of Queries	Time (ms)	No. of Objects Faulted
Original	51	836	63
Lazy	55	502	60
AutoFetch	52	522	60

Table 2.4: Comparison of Roller versions with respect to the number of queries, time, and the number of objects faulted into memory when loading the page for a blog entry.

Version	No. of Queries	Time (ms)	No. of Objects Faulted
Original	47	664	49
Lazy	52	360	46
AutoFetch	46	363	46

for all queries; so the existing prefetch policy is a compromise developed by the programmers that works well for most queries. Interestingly, the original version executes less queries than the lazy version, but is much slower. This is due to redundant prefetch directives that make the queries more complicated than necessary and the additional object loads.

Table 2.4 shows the number of queries, time, and number of objects faulted for each version of the Roller application while visiting the comments page for a single blog entry. There are four comments. Again the AutoFetch performs as well as the lazy version while executing fewer queries.

2.5.5 Ebean Website

Independently of our implementation of AutoFetch, Rob Bygrave incorporated AutoFetch into Ebean, an ORM tool. Ebean supports programmatic queries similar to Hibernate Criteria queries. Unlike our implementation, Ebean AutoFetch tracks access for all properties of an object including primitive values. This information can be used to prefetch individual object properties producing partial objects in memory. By default, Ebean loads all of the primitive fields of an object. Using the profile information, Ebean AutoFetch optimizes the loading of objects to only include the fields required by the program. Thus in this case, AutoFetch is used to load less data than the default query strategy.

The Ebean website is built on top of the Ebean ORM using AutoFetch. It contains several modules: a forum, bug tracker, user account management, and internal query statistics. This last module displays statistics on all queries executed on the website including AutoFetch profiles. For the rest of this section, we consider a snapshot of these statistics taken on May 30th, 2009.

Of the top ten queries taking the most aggregate time, 8 of 10 queries are modified by AutoFetch. Of these eight queries, six queries involve prefetch for object associations and the other two queries only optimize the properties loaded for objects. In total, 40 query classes are tracked by AutoFetch for the website. For all the query classes but one, the traversals are deterministic, i.e. the traversals are the same for each query invocation.

Figure 2.12 shows the distribution for the number of association prefetches added to queries which corresponds to the number of association traversed. The number of prefetches ranges from zero to seven with most queries having between zero and three prefetches added. Figure 2.13 shows the distribution for the depth of association prefetches which corresponds to the depth of the traversal. Unlike the OO7 benchmark which has traversals that are up to 29 associations deep, the Ebean website never prefetches more than two levels deep and the majority of prefetches are one level deep. This matches our hypothesis that in many enterprise applications the query traversals are shallow. The query statistics on the website do not give any information on the breadth of the traversals, i.e. the number of objects that returned by queries and in collection associations.

2.5.6 General Comments

In all of the evaluation benchmarks, the persistent data traversals were the same given the query class. Consequently, AutoFetch never prefetched more data than was needed, i.e. AutoFetch had perfect precision. While our intuition is that persistent data traversals are usually independent of the program branching behavior, it is an open question whether our benchmarks are truly representative in this respect. Similarly, it is difficult to draw general conclusions about the parameters of the AutoFetch such as the maximum extent level or stack frame limit without observing a larger class of persistent programs. The maximum extent level was set to 12, because this produced

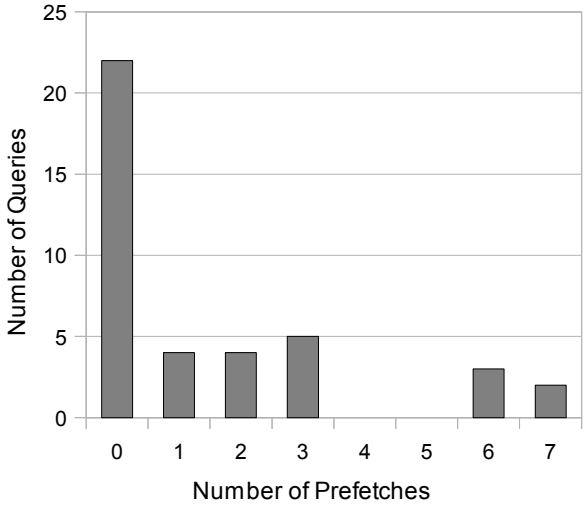

Figure 2.12: The distribution for size of traversals in Ebean.

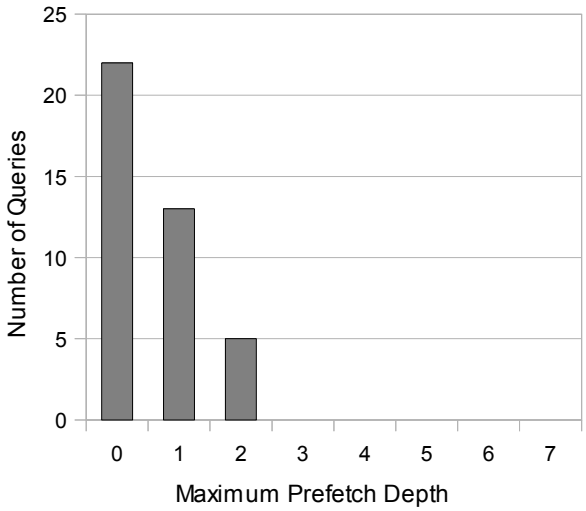

Figure 2.13: The distribution for depth of traversals in Ebean.

reasonable memory consumption on our benchmarks. The stack frame limit was set to 20 to preserve enough information from the stack frame about control flow in the presence of the various architectural layers in the Torpedo benchmark, the recursive traversals in the OO7 benchmark, and the indirection in J2EE applications.

AutoFetch performed the worst on the OO7 benchmarks because of the depth of the traversals and because some of the traversals were on a strongly connected graph. Consequently our hypothesis that the number of queries is the only important metric is only partially true. The complexity of queries in terms of the number of joins, the size of the SQL query results, and the overhead of dynamic profiling can be an issue for some traversals. The strength of AutoFetch lies in addressing the $n+1$ select problem as seen in the Torpedo benchmark. In this case, a single join can preclude the execution of many subsequent queries.

2.6 Related Work

Han et al. [76] classify prefetching algorithms into five categories: manually specified prefetches, page-based prefetching, object-level/page-level access pattern prefetching, context-based prefetches, and traversal/path-based prefetches. We add to those categories structure-based prefetch.

Most ORM tools allow programmer to manually specify prefetch. The prefetch may be specified at the type-level by annotating class associations that should always be faulted in with objects of that class. Or the prefetch

may be specified in explicit queries. Thor [99, 56] allows objects to be grouped such objects in the group are faulted together. Thor also incorporated some other prefetch strategies discussed in Section 2.6.3

2.6.1 Page-based Prefetching

Page-based prefetching has been explored in object-oriented databases such as ObjectStore [96]. A page is a continuous region on disk which contains multiple objects. In page-based prefetch, pages are faulted into memory instead of objects. Thus, accessing multiple objects on a single page requires only one fault. The size of a page is chosen to match characteristics of the persistent hardware, allowing for very low-cost prefetch. Page-based prefetching is effective when the access patterns of an application correspond to the clustering of the objects on disk. Since the clustering is usually static, it cannot efficiently support multiple data access patterns. Good clustering of objects is difficult to achieve and can be expensive to maintain when objects are updated frequently. Poor clustering leads to the program loading many more objects into memory than the program requires without any of the benefits of prefetch.

2.6.2 Object-level Prefetching

Object-level pattern prefetching relies on monitoring the sequence of object or page requests to the database. Object-level pattern prefetching relies on recognizing patterns in object faults and using that information to insert prefetch directives. Page-level patterns may be used instead of object-level

patterns which produces a mixture of object-level pattern prefetching and page-based prefetch.

Palmer and Zdonik [115] implemented a prefetch system, Fido, that stores access patterns and uses a nearest neighbor algorithm to detect similar patterns and issue prefetch requests.

Curewitz et al. [51] implemented an access pattern prefetching algorithm using compression algorithms. Their insight is that compression algorithms impose (implicitly or explicitly) a dynamic probablity distribution over the input data. This dynamic probability distribution can be used to insert prefetch directives. A lot of their work is focused on how prefetching interacts with the local cache. In contrast, we make two assumptions:

- Caching occurs only within a transaction. Because relational databases are often shared, caching outside a transaction is complex and dependent on application semantics.

- All the objects needed by a transaction fit into memory. This is a good assumption for most OLTP scenarios because transactions are short-lived. Most ORM tools including Hibernate depend on this assumption to maintain object reference equality; all the objects touched in a transaction are stored in a session cache. Hibernate does offer some limited support for bulk transactions through a separate API.

Knafla [93] models object relationship accesses as discrete time Markov chains and uses this model in addition to a sophisticated cost model to issue

prefetch requests. The discrete time Markov chain models can be computed offline using LU factorization or online using conjugate gradient. If the model is computed online, a separate thread continuously updates the model. He implemented his system for the Exodus object database which uses page-level faulting.

The main drawback to these approaches is that they detect object-level patterns, i.e. they perform poorly if the same objects are not repeatedly accessed. As the database size grows and object associations change, it becomes more unlikely that object-level patterns are sufficient. We found that repeated access to the same objects is not typical of many enterprise applications because transactions tend to access independent sets of data.

2.6.3 Structure-based Prefetch

Day [56] examined different prefetch strategies for the Thor system. Most of these were a form of structure-based prefetch; prefetching objects that are connected to loaded objects in the object graph. He tested the different strategies on OO7 and best performing strategy was breadth-first with cutoff. In the breadth-first with cutoff strategy, each time an object is loaded, the database management system performs a breadth-first traversal of the object graph starting from the requested object. Once the breadth-first traversal visits a number of objects determined by the cutoff parameter, all the objects visited in the traversal are sent to the client program. The main disadvantage of this prefetching strategy is that it is relatively imprecise; it brings all related

objects into memory instead of trying to predict which ones will be needed by the program.

2.6.4 Context-based Prefetching

Context-based prefetching is similar to structure-based prefetch in that prefetch objects based on their associations; however, context-based prefetch uses information about the program execution to guide the prefetch. Bernstein et al. [18] proposed a context-controlled prefetch system, which was implemented as an extension of Microsoft Repository. Each persistent object in memory is associated with a context. This context represents a set of related objects, either objects that were loaded in the same query or objects that are a member of the same collection association. For each attribute access of an object O, the system prefetches the requested attribute for all objects in O's context. When iterating through the results of a query or collection association, this prefetch strategy will avoid executing $n + 1$ queries where n is the number of query results or the size of the collection. A comparison of this strategy and AutoFetch is given in Section 2.6.5. While AutoFetch only profiles associations, Bernstein et al. use "MA prefetch" to prefetch scalar attributes for classes in which the attributes reside in separate tables. MA prefetch improves the performance of the OO7 benchmark queries, which were not improved by AutoFetch, because OO7 attributes and associations are separated into multiple tables. The implemented context-controlled prefetch only supported single-level prefetches, although prefetching multiple

levels (path prefetch) is mentioned as future work. The system also makes extensive use of temporary tables, which are not needed in AutoFetch.

2.6.5 Path-based Prefetching

Han et al. [77, 76] extended the ideas of Bernstein et al. to maintain not only the preceding traversal which led to an object, but the entire type-level *path* to reach an object. In our terminology, these paths are defined on the type graph. Each query is associated with an attribute access log set which contains all the type level paths used to access objects from the navigational root set. The prefetch system then monitors the attribute access log and prefetches objects if either an iterative or recursive pattern is detected. The prefetch system, called PrefetchGuide, can prefetch multiple levels of objects in the object graph if it observes multi-level iteration or recursive patterns. However, unlike the Bernstein prefetch implementation, prefetch is is only added to navigational queries. PrefetchGuide is implemented in a prototype ORDBMS.

While the systems created by Bernstein and Han prefetch data within the context of a top-level query, AutoFetch uses previous query executions to predict prefetch for future queries. Context-based prefetch always executes at least one query for each distinct association path. AutoFetch, in contrast, can modify the top-level query itself, so that only one query is needed. AutoFetch can also detect traversal patterns across queries, e.g. if certain unrelated associations are always accessed from a given query result,

AutoFetch prefetches those objects even though it would not constitute a recursive or iterative pattern within that single query. One disadvantage of AutoFetch is that the initial queries are executed without any prefetch at all. The consequence of this disadvantage, is that the performance on the initial program iteration is equivalent to a program with unoptimized queries. However, it would be possible to combine AutoFetch with a system such as PrefetchGuide. In such a combined system, PrefetchGuide could handle prefetch in the first query, and also catch cases where the statistical properties of past query executions do not allow AutoFetch to predict correct prefetches. We believe that such a combination would provide the best of both worlds for prefetch performance.

2.6.6 Local Memory Prefetch

Automatic prefetch in object persistence architectures is similar to prefetching data from memory into the L1 or L2 cache. A number of papers have examined software approaches.

Luk and Mowry[102] have looked at optimizing recursive data structure access by predicting which parts of the structure will be accessed in the future. They use history pointers as part of a dynamic analysis to predict a memory access n iterations later. They also use data linearization to map recursive data structures to an array so that a prefetch address can be computed using constant offsets.

Inagki and al. [85] implemented a just in time compiler optimization in

a Java vitual machine for stride prefetching. Stride prefetching is a particular prefetch pattern in which a memory access has a constant offset with respect to a pointer. For example, accessing a field of a structure or object. The stride prefetches were determined using a intra-procedural analysis which takes into account runtime values.

Marathe and Mueller built PFetch [104] which instruments programs to construct a sequence of timestamped memory accesses. These memory addresses are used in an offline analysis which simulates the memory and cache behavior and find potential profitable prefetch instructions. Their prefetch model supports prefetching a constant offset from a pointer or a pointer value which supports a wide range of prefetch patterns including stride prefetching.

There are also numerous hardware prefetching approaches which implement many of the ideas above in hardware. In constrast to these systems, AutoFetch's analysis is online, adaptive, inter-procedural and supports all the patterns of prefetching described by Marathe and Mueller. The key is that we take advantage of some of the differences between persistent data prefetch in enterprise applications and memory prefetch:

- Precision is more important in memory prefetch. The cache is not large enough to contain the entire working set of the program and there is no unit of work such as the transaction in persistent programs.
- Coverage is more important in persistent data prefetch. Because the persistent object cache is often large enough to hold all the objects in a

single transaction, the program can profitably prefetch a lot of data. For memory prefetch, too much prefetch can degrade other memory accesses because they share the same cache.

- The amount of computation in memory prefetch is limited. Because object persistent faulting is several orders of magnitude slower than memory prefetch, there is more latitude for computation such as examining the stack frame.

For these reasons, we do not think AutoFetch would translate well to memory prefetch, although an exploration of that would be interesting.

2.6.7 Distributed Memory Prefetch

Distributed memory access is more similar to persistent object prefetch because they both incur the overhead of network communication. The inspector-executor compilation strategy [94] allows the optimization of remote memory accesses. For each parallel loop, the compiler generates inspector and executor code. In the inspector phase, the program examines remote accesses and determines the data dependencies between them. In the executor phase, the program executes the remote memory accesses while optimizing the communication schedule. Most of the inspector-executor optimizations are focused on array data structures For example, Yokota et al. [162] took advantage of support for stride prefetching in their remote memory infrastructure to batch together remote memory access which were at regular intervals.

Viswanathan et al. [153] explored a similar static analysis in the context of a predictive cache coherence protocol and mutable pointer-based data structures. A static compiler analysis prefetches data from remote collections (referred to as aggregates) for each iteration in a parallel loop.

Unlike AutoFetch, there are no explicit queries when accessing distributed memory, rather the program requests a single memory location at a time.

2.7 Future Work

We presented a simple query classification algorithm which only relies on the call stack at the moment the query is executed. Although we found this to work quite well in practice, a more complex classification algorithm could include other features of program state: the exact control path where the query was executed, or the value of program variables. This richer program state representation might classify queries too finely. Unsupervised learning techniques could be applied to richer program state representations to learn a classification that clusters the queries according to the similarity of their traversals. Consider the following program fragment, where `findAllFoos` executes a query:

```
List results = findAllFoos();
if (x > 5)
  doTraversal1(results);
else
  doTraversal2(results);
```

A learning algorithm could learn a better classification strategy than the one described in this paper. In this case, the value of the variable `x` should be used to distinguish two query classes.

A related point is finding a way to more efficiently compute the query class for a program point. The current implementations use a stack-trace obtained from the JVM that is fairly expensive to compute because it traverses up the call stack to collect all the calling methods. Another option is to use PCC [27] which can compute a probabilistic approximation of the call stack with low overhead (3%).

A cost model for database query execution is necessary for accurate optimization of prefetching. AutoFetch currently uses the simple heuristic that it is always better to execute one query rather than two (or more) queries if the data loaded by the second query is likely to be needed in the future. This heuristic relies on the fact that database round-trips are expensive. However, there are other factors that determine cost of prefetching a set objects: the cost of the modified query, the expected size of the set of prefetched objects, the connection latency, etc. A cost model that takes such factors into account will have better performance and may even outperform manual prefetches since the system would be able to take into account dynamic information about database and program execution.

2.8 Conclusion

Object prefetching is an important technique for improving performance of applications based on object persistence architectures. Current architectures rely on the programmer to manually specify which objects to prefetch when executing a query. Correct prefetch specifications are difficult to write and maintain as a program evolves, especially in modular programs. AutoFetch is a novel technique for automatically computing prefetch specifications. AutoFetch predicts which objects should be prefetched for a given query based on previous query executions. AutoFetch classifies queries executions based on the client state when the query is executed, and creates a traversal profile to summarize which associations are traversed on the results of the query. This information is used to predict prefetch for future queries. Before a new query is executed, a prefetch specification is generated based on the classification of the query and its traversal profile. AutoFetch improves on previous approaches by collecting profile information across multiple queries, and using client program state to help classify queries. We evaluated AutoFetch using both sample applications and benchmarks and showed that we were able to improve performance and/or simplify code.

Chapter 3

RBI-DB

One of the key issues we are trying to solve when using transparent persistence is optimizing the number of queries sent to the database. We assume that the database is located in a different process on the same machine or on a different machine altogether. Reducing the number of round-trips to a remote server is a fundamental problem in distributed computing. For most of this chapter we address the general problem of how to batch remote procedure calls in Java. Then at the end of the chapter, we show how transparent persistence can be mapped to remote procedure calls and how to translate a batch of remote procedure calls into a SQL query.

The Remote Procedure Call (RPC) has long been the foundation of language-level approaches to distributed computing. The idea is simple: replace local calls with stubs that transfer the procedure call to a remote machine for execution. RPC has been generalized for objects to create distributed object systems, including Common Object Request Broker Architecture (CORBA) [114], the Distributed Component Object Model (DCOM) [30], or Java Remote Method Invocation (RMI) [135]. Stubs are defined on a local object that acts as a proxy for a remote object. One advantage of this ap-

proach is that it does not require language changes, but can be implemented using libraries and stub generator tools.

Standard object-oriented designs, which focus on flexibility and extensibility through the use of fine-grained methods, getters and setters, and small objects, do not perform well when distributed remotely. Every method call on a remote proxy is a round trip to the server. To achieve suitable performance, remote objects must be designed according to a different set of principles[1]. Data Transfer Objects and Remote Façades are used to optimize data transfer and combine operations to reduce the number of round trips [66]. One effect of this approach is that servers and protocols are hard-coded to support specific client invocation patterns. If a client changes significantly, then the entire system, including the server and its interfaces, must be redesigned.

Remote Batch Invocation (RBI) is a new approach to distributed object computing. Remote Batch Invocation allows multiple calls on remote objects to be invoked in a batch, while automatically transferring arguments and return values in bulk. The following example uses a Remote Batch in Java to delete low-rated albums from a personal online music database.

```
int minimum = 5;
Service musicService = new Service("MusicCloud", Music.class);
batch (Music favoriteMusic : musicService) {
  for (Album album : favoriteMusic.getAlbums())
    if (album.rating() < minimum) {
      System.out.println("Playing: " + album.getTitle());
      try {
```

[1]Approaches using asynchronous messaging are discussed in related work

```
        album.play();
      } catch (Exception e) {
        System.out.println("error: " + e.getMessage());
}}}
```

The batch mixes local and remote computation. In this case, all the computation is remote except the two calls to `System.out`. The semantics of Java is modified within the batch to first perform all remote operations, then perform all local operations. Thus the typical ordering between local and remote statements is not necessarily preserved. For example, all of the albums are played before any of the names are printed. All loops and conditionals are executed twice: once on the server and then again on the client. Exceptions on the server terminate the batch by default, and raise the error in the analogous execution point on the client.

A remote batch transfers all data between client and server in bulk. In this case, just the `minimum` rating is sent to the server. The server returns a list of all titles of played albums. But it also returns a boolean for each album indicating whether it was played. In general, any number of primitive or serializable values can be transfered to and from the server. Remote Batch Invocation creates appropriate Data Transfer Objects and Remote Façades on the fly, involving any number of objects and methods. Standard Java objects can be published as a batch service by adding a single line of code. The semantics of the batch statement require that only a single remote invocation is made in the lexical block. This strong performance model is important, because the cost of remote invocations may be several orders of magnitude

higher than local invocations.

We demonstrate Remote Batch Invocation with an extension to Java. A source-to-source translator converts the `batch` statement to plain Java. Remote Batch Invocation is not tied to RMI, but could also be implemented using other middleware transport, for example web services or mobile objects. A server can publish a remote service by making a single library call.

The performance benefits of batching operations are well-known, especially in high-latency environments. We evaluate our language extension by comparing it with other approaches to batching such as implicit batching, mobile code, and the Remote Façade pattern.

In summary, Remote Batch Invocation is a new approach to distributed objects that supports service-orientation rather than remote procedure calls and proxies. The fundamental insight is that remote execution need not work at the level of procedure calls, but can instead operate at the level of blocks, with bulk transfer of data entering and leaving the block. Unlike traditional distributed objects that maintain server side state, Remote Batch Invocation has a stateless execution model that is characteristic of service oriented computing [95, 63].

3.1 Remote Batch Invocation

Remote Batch Invocation allows clients to combine remote operations into a single remote invocation. We will illustrate the features of Remote Batch

Invocation by example. The basis of our examples is a sample remote service described by Fowler in *Patterns in Enterprise Application Architecture* [66]. This simple remote music service is comprised of three classes: Album, Artist, and Track as shown in Figure 3.1. The `Album` interface also provides the `play` method which returns the lyrics on the album and plays the album on a sound system.

```
interface Album {
  String getTitle();
  void setTitle(String title);
  Artist getArtist();
  void setArtist();
  Track[] getTracks();
  void addTrack(Track t);
  void removeTrack(Track t);
  String play();
}
```

A natural remote interface to these three classes is shown below:

```
interface Music {
  Album createAlbum(String id, String title);
  Album getAlbum(String id);
  Artist addArtist(String id, String name);
  Artist getArtist(String id);
  Track createTrack(String title);
}
```

Using the `Music` interface, a client can create and find artists and albums as well as create tracks. A client may update object fields using the appropriate setters. We will use this interface for our Remote Batch Invocation examples.

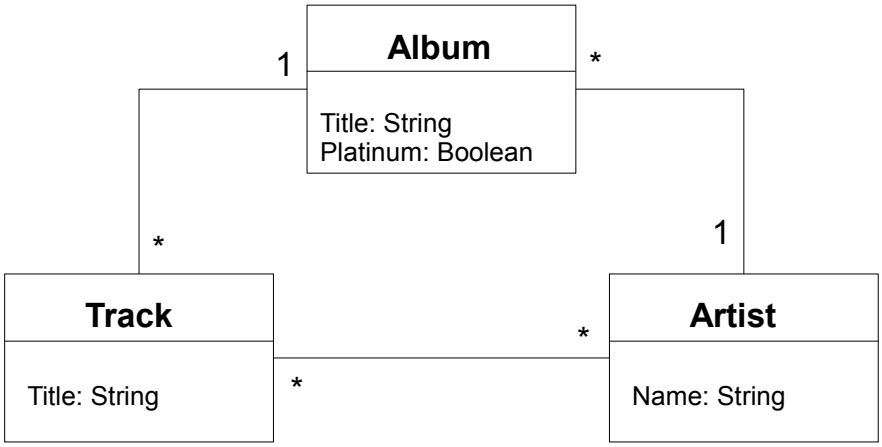

Figure 3.1: UML class diagram for Fowler album data model.

Unfortunately, this natural interface is too fine-grained in a system where individual method calls are expensive. Using the Remote Façade and Data Transfer patterns, Fowler wraps the Music interface:

```
interface FowlerMusic {
  String play(String id);
  AlbumDTO getAlbum(String id);
  void createAlbum(String id, AlbumDTO dto);
  void updateAlbum(String id, AlbumDTO dto);
  void addArtistNamed(String id, String name);
  void addArtist(String id, ArtistDTO dto);
  ArtistDTO getArtist(String id);
}
```

`FowlerMusic` is a Remote Façade for the `Music` interface. For example, the `FowlerMusic.play` method is simply calling the `Music.getAlbum` method followed by the `Album.play` method. The `AlbumDTO`, `ArtistDTO`, and `TrackDTO` are data transfer objects (DTO) that transfer information in bulk to and from

the remote server. Fowler also defines `AlbumAssembler`, which maps between DTOs and objects residing on the server.

```
class AlbumAssembler {
  public AlbumDTO writeAlbum(Album subject) {
    AlbumDTO result = new ALbumDTO();
    result.setTitle(subject.getTitle());
    result.setArtist(subject.getArtist().getName());
    writeTracks(result, subject);
  }
  void writeTracks(AlbumDTO result, Album subject) { ... }
  void writePerformers(TrackDTO result, Track subject) { ... }
  public void createAlbum(String id, AlbumDTO source) {
    Artist artist = Registry.findArtistNamed(source.getArtist());
    if (artist == null) throw new RuntimeException(...);
    Album album = new Album(source.getTitle(), artist);
    createTracks(source.getTracks(), album);
    Registry.addAlbum(id, album);
  }
  void createTracks(TrackDTO[] tracks, Album album) { ... }
  void createPerformers(Track newTrack, String[] performers) { ... }
}
```

Although `AlbumAssembler` encapsulates the logic of mapping between DTO and model objects, it is not generic, containing a hard-coded decision about the DTO content. In the book, Fowler decides to have the Album DTO provide all the information about a single album.

The next sub-sections give examples of using Remote Batch Invocation for batch data retrieval, batch data transfer, loops, branching, and exceptions.

3.1.1 Batch Data Retrieval

A simple client may want to print the title and name of the artist for an album. With the fine-grained `Music` interface, the client must execute four remote calls: a call to find the album, a call to get the title of the album, a call to get the artist for the album, and a call to get the name of the artist for the album.

Using Remote Batch Invocation, the client can use the `Music` interface while still executing a single remote call. The input to the remote batch is the id of the album "1". The output of the remote batch is the title of the album and the name of the artist of the album. A remote batch can combine an arbitrary number of method calls as long as they are invoked on objects transitively reachable from the root object of the batch, in this case `music`.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Title: " + album.getTitle());
  System.out.println("Artist: " + album.getArtist().getName());
}
```

The same client using the remote façade `FowlerMusic` executes a single remote method `getAlbum` which returns `AlbumDTO`. For this client, the DTO is an over-approximation of the data needed; a Remote Façade optimized for this client would need another DTO for albums that only provides the title and artist name.

```
AlbumDTO album = music.getAlbum("1");
System.out.println("Title: " + album.getTitle());
System.out.println("Artist: " + album.getArtistName());
```

For other clients, the DTO may be an under-approximation of the data needed. For example, this client prints the title of two different albums.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Title: " + album.getTitle());
  final Album album = music.getAlbum("2");
  System.out.println("Title: " + album.getTitle());
}
```

`FowlerMusic` does not contain a method that matches this client pattern. Consequently, the same client using `FowlerMusic` must make an additional remote call compared to using Remote Batch Invocation. Alternatively, the `FowlerMusic` interface can be changed to include a method that takes two album IDs as input and returns a new DTO containing two fields representing the titles of the input albums. This highlights one of the disadvantages of the Remote Façade pattern; it creates a non-functional dependency between the server interface and the client call patterns.

3.1.2 Batch Data Transfers

Remote Batch Invocation also allows clients to transparently transfer data in bulk to the server. The following code creates `Album`, `Artist`, and `Track` objects and wires them together. The input to the remote batch is all the information about the album, artist, and track to be created and there is no output. The actual construction of the objects and method calls occur entirely on the server.

```
batch (Music music : musicService) {
```

```
  final Album album = music.createAlbum("2", "First Album");
  final Artist artist = music.addArtist("2", "John Smith");
  album.setArtist(artist);
  final Track track = music.createTrack("First track");
  track.addPerformer(artist);
  album.addTrack(track);
}
```

A client using `FowlerMusic` can also create the objects using a single remote invocation using the appropriate DTOs.

```
AlbumDTO album = new AlbumDTO("First Album");
AlbumDTO artist = new ArtistDTO("2", "John Smith");
album.setArtist(artist);
TrackDTO track = new TrackDTO("First Track");
track.addPerformer(artist);
album.addTrack(track);
music.createAlbum("2", album);
```

A drawback to using data transfer objects for creating and updating objects, is that DTO is under-specifying some of the semantics of the operation. In particular, the DTO does not tell the server whether the artist object is an artist object which should be created or if it already exists. This is a well-known problem in data mapping and commonly arises in distributed systems. A common approach and the one taken by Fowler in his book, is to specify a convention to either always create objects, always use existing objects, or create an object if it does not already exist. Another approach is to enrich the DTO with *status* fields for each normal field that specify the right semantics. Sometimes this status field is encoded into the field, for example, by using `null` as a special value. A related problem is updating objects if the client

only has a partial description of the object. The client must be able to update the subset of fields which are known, but not the fields which are unknown.

The remote batch is more explicit in that specifies that the `artist` is a new `Artist` object. If the client wanted to reference an existing artist the code would be rewritten as follows:

```
batch (Music music : musicService) {
  final Album album = music.createAlbum("2", "First Album");
  final Artist artist = music.getArtist("2");
  album.setArtist(artist);
  final Track track = music.createTrack("First track");
  track.addPerformer(artist);
  album.addTrack(track);
}
```

3.1.3 Loops

So far, we have shown that Remote Batch Invocation supports straight-line code. However, it is common for a client to need more complex logic involving branching and loops. Remote Batch Invocation allows for remoting of the enhanced `for` loop introduced in Java 1.5 if the collection can be evaluated remotely. If data from the iterations is needed locally, the remote batch constructs a data transfer object with an array of the data needed and transparently maps it on the client. Below is a simple example which shows how explicit batching can operate over arrays. The input to the remote batch is simply the id of the album and output is the title of all of the tracks, the name of all of the performers on the tracks, and the lyrics returned by the `play` method.

```
batch (Music music : musicService) {
  final Album album = music.getAlbum("1");
  System.out.println("Tracks: ");
  for (Track t : album.getTracks()) {
    System.out.print(t.getTitle());
    System.out.println(',');
    System.out.print("Performed by: ");
    for (Artist a : t.getPerformers()) {
      System.out.print(a.getName());
      System.out.print(' ');
    }
    System.out.print('\n');
  }
  System.out.println("Song: " + album.play());
}
```

The `FowlerMusic.getAlbum` method in Remote Façade nearly provides all the functionality required by this client; however, it does not include a call to the `Album.play` method.

3.1.4 Branching

Conditional statements, including `if` and `else`, are remoted if their condition is a remote operation. Below is a simple example that shows such a remoted conditional statement also containing the primitive operator `&&`.

```
batch(Music music : musicService) {
  final Album album = registry.getAlbum("1");
  if (album.getName().startsWith("A")
       || album.getName().startsWith("B")) {
    album.play();
    System.out.print("Title starts with A or B: " + album.getTitle());
  } else {
    System.out.print("Title does not start with A or B: "
```

```
      + album.getArtist().getName());
  }}
```

RBI supports boolean and numeric primitive operators, both unary and binary. Conditional code can also be included as part of operations on collections. In that case, the conditions are reevaluated on each iteration over a collection. The following example adds albums composed by Yo-Yo Ma to the favorites collection.

```
for (Artist a : t.getPerformers()) {
  if (a.getName().equals("Yo-Yo Ma")) {
    favorites.addArtist(a);
  }}
```

3.1.5 Exceptions

Remote Batch Invocation separates exceptions caused by failures in communication from logical exceptions that arise when executing the statements in the batch. The `batch` statement itself can raise network exceptions, which must be handled by the surrounding context. If there are no network errors, then exceptions raised by statements in the batch can be handled in the client.

Within a `batch`, a remote operation can raise an exception on the server that will terminate the batch. The thrown exception will be raised in the corresponding execution point on the client. The client must use exception handlers as in regular Java code. In addition, the execution of a remote batch may result in a `RemoteException` that can be handled by wrapping an entire

`batch` block with a `try/catch` block.

For example, the following code extends an earlier example to include an exception handler when trying to play an album, and another handler that deals with network and communication errors raised at any point of executing the batch.

```
try {
  batch (Music favoriteMusic : musicService) {
  ...
    try {
      album.play();
    } catch (PermissionError pe) {
      System.out.println("No permission to play album"
        + album.getTitle());
    }
  } //end batch
} catch (RemoteException re) {
  System.out.println("Error communicating batch.");
}
```

The default behavior of a batch is to abort processing when an exception is thrown. As future work, we would like to be able to apply a different exception policy, for example to continue execution or restart the batch. Batches also provide a natural unit of atomic execution. In many cases it is desirable for the entire batch to succeed or fail, so that incomplete operations are never allowed. One way to achieve this is to use transactional memory on the server [29].

Even so, it is possible for the batch to succeed on the server but for a communication error to prevent the client from completing the batch. A

standard two-phase commit could be used to ensure that both the server and client parts of the batch have executed to completion. These topics are beyond the scope of our current research, but we do not see any obstacles to combining RBI with distributed transactions.

3.1.6 Service Implementation

Implementing a Remote Batch Invocation service is much simpler than implementing a server using traditional distributed object middleware, including RMI or CORBA. There is no need to create method stubs. Instead, the server simply registers a root object with a single call after creating the server implementation object.

```
Music musicServer = new MusicImpl(...);
rbi.Server server = new rbi.Server("MusicCloud", musicServer);
```

The client connects to this service by using the same name and interface.

```
rbi.Service musicService =
  new rbi.Service("MusicCloud", Music.class);
```

As in most distributed systems, interface mismatches between client and server are detected at runtime. Standard Java interfaces define the service contract.

3.1.7 Service-Oriented Interaction

Remote Batch Invocation supports a service-oriented style of interaction, so it does not support object proxies. This is not a problem for many client/server interactions, which can be naturally accomplished in a single round-trip. These interactions have the following pattern:

client $\overset{input}{\longrightarrow}$ server* $\overset{results}{\longrightarrow}$ client

The client sends any number of inputs to the server, which performs multiple actions and returns any number of results to the client. There may be cases; however, when a server computation depends upon client input *and* previously defined server objects.

client $\overset{input}{\longrightarrow}$ server* $\overset{results}{\longrightarrow}$ client* $\overset{input_2}{\longrightarrow}$ server* $\overset{results_2}{\longrightarrow}$ client

This situation is easily handled in distributed object systems like CORBA and RMI, since each server operation is controlled by the client and it can use proxies to refer to the intermediate server results needed in the last step.

This interaction pattern requires some other solution in a stateless service-oriented system. The simplest approach is to have the second server batch reload or recreate the server objects that were defined in the first batch. The server may also provide public identifiers for its objects. The first *results* can include a server object identifier, which is used in the second batch to relocate the necessary server object. These patterns have been studied extensively in the context of service-oriented computing [95, 63].

3.1.8 Allowed Remote Operations

Any Java code may appear inside the batch block; however, the compiler enforces some data flow restrictions described in Section 3.2. Many Java constructs such as constructor calls, casts, `while` loops, and assignments can-

not be remoted; they are always executed on the client. Future work may relax some of these restrictions. If remote assignments were allowed, then it would be possible to aggregate (e.g. sum or average) over collections remotely. General loops could also be remoted without significant changes to the model.

Exceptions are a special case. The remote batch cannot catch exceptions remotely, but it does propagate them to the client in the original location of the remote operation that produced the exception. In this way, the client can catch exceptions raised remotely and handle them locally.

Keeping the remoteable constructs simple and as universal as possible increases the viability of using RBI against remote interfaces written in other languages.

3.2 Semantics

Our Java implementation of Remote Batch Invocation uses the following syntax:

`batch (`*`Type Identifier : Expression`*`) Block`

The *Identifier* specifies the name of the root remote object. The *Expression* specifies the service which will provide the root remote object. The *Block* specifies both remote and local operations. A remote operation is an expression or statement executed on the server. A batch block is *partitioned* into a remote block containing all the remote operations and a local block con-

taining the local operations. Within each block, the operations are executed in same relative order as in the batch block. The remote block is executed first followed by the local block, so the local block may depend on data produced by the remote block, but not vice-versa. The remote block may take as parameters constants and variables not modified in the batch block. Data is communicated between the two blocks by value.

The partition operation can be thought of as the dual of the *deforestation* [155]. Deforestation is an optimization technique for combining two method calls into a single method call and eliminating intermediate tree data structures. Our partitioning splits the block and creates the appropriate intermediate data structure needed to pass information between the blocks. The remote block is executed first with a **single** remote call using remote evaluation [131], a form of mobile code. This is a key non-functional property as it provides a strong performance model to the programmer, albeit lexically scoped.

Exceptions in a remote operation are re-thrown in the local operation sequence at the original location of the remote operation. If the remote operations fail due to a network error, then an exception is thrown before any of the local operations execute. Operations inside the batch block are reordered during the partitioning and it is possible that the block executes differently as a batch than it normally would. The compiler does try to identify some of these cases and warn the programmer, however, it is up to the programmer to be aware of the different Java semantics inside the batch block.

$$
\begin{aligned}
n &\in Name \\
l &\in Variable \\
v &\in Value \\
binop &\in \{+, -, *, /, \vee, \wedge, >, =\} \\
unop &\in \{-, not\} \\
E &= \texttt{root} \mid l \mid E\ binop\ E \mid unop\ E \mid v \\
&\quad \mid E.m(E_1, \ldots, E_n) \\
S &= E \\
&\quad \mid ; \\
&\quad \mid S_1; S_2 \\
&\quad \mid \texttt{let}\ l = E\ \texttt{in}\ S \\
&\quad \mid \texttt{let*}\ l = E\ \texttt{in}\ S \\
&\quad \mid \texttt{if}\ S_1\ S_2\ S_3 \\
&\quad \mid \texttt{for}\ (l \in E)\ S
\end{aligned}
$$

Figure 3.2: Domain specific language for remote block operations

The remote code in a remote partition is restricted. Figure 3.2 shows the domain specific language that can express the allowed operations in a remote block.

The language contains basic control structures for sequencing, naming, branching, and looping. The `let*` construct behaves similar to the normal binding `let` construct, but additionally marks that binding as required in a local operation. The keyword `root` represents the root service object. The values in this language are any Java value. The language includes some simple arithmetic and logical operators to operate over the primitive types. General assignment is not supported in the remote partition. Therefore, variables are

only remote if they correspond to the `batch` variable or if they are `final` and assigned remote expressions.

The partitioning algorithm identifies expressions and statements as *local* or *remote*. Local expressions are further subdivided into *static locals* and *non-static locals*. Remote expressions and statements execute on the server, possibly with input from static local expressions. Local expressions and statements execute on the client, possibly with output from remote expressions. Static local expressions are literals and variable expressions defined outside of the batch and not assigned within the batch before their use. All other local expressions are non-static. Java 1.5 `for` statements are executed remotely if their collection is a remote expression. A remote `for` loop is duplicated in the local partition to support local expressions or statements inside the loop. Similarly, conditional statements are executed remotely if their condition is a remote expression. A remote conditional is duplicated in the local partition to support local expressions or statements inside the `if` statement. All other statements are executed in the local partition, e.g. `while` loops.

3.2.1 Expression Locations

The compiler determines the location of an expression statically. A component of this analysis is a forward flow-sensitive data-flow analysis that maps variables to locations. Locations are ordered as a small lattice where *static local* $<$ *remote* $<$ *non-static local*. The $\uplus$ operator adds or changes a mapping for a variable. The *pred* function returns the predecessors of a

$$
\begin{aligned}
n, m &\in \mathit{Statement} \\
e &\in \mathit{Expression} \\
\mathit{inBatch}(e) &= \begin{cases} \mathit{true} & \text{e is inside batch statement} \\ \mathit{false} & \text{otherwise} \end{cases} \\
\mathit{varBatch}(e) &= \begin{cases} v & \text{e inside batch statement of the form } \mathsf{batch}(T\ v : e) \\ \mathit{undefined} & \text{otherwise} \end{cases} \\
s \uplus \mathsf{nil} &= s \\
s \uplus [v \mapsto l] &= \begin{cases} s \cup [v \mapsto l] & [v \mapsto \_] \notin s \\ (s - [v \mapsto k]) \cup [v \mapsto l] & [v \mapsto k] \in s \end{cases} \\
\mathit{in}[n] &= \bigcup_{m \in \mathit{pred}(n)} \mathit{out}[m] \\
\mathit{out}[n] &= \mathit{in}[n] \uplus \mathit{gen}(n) \\
\mathit{gen}(n) &= \begin{cases} [v \mapsto \text{remote}] & n = [\![\mathsf{batch}(T\ v : e)]\!] \\ [v \mapsto \text{static local}]) & n = [\![v = e]\!] \wedge\ !\ \mathit{inBatch}(n) \\ [v \mapsto \text{non-static local}] & n = [\![v = e]\!] \ \wedge\ \mathit{varBatch}(n) \neq v_b \\ [v \mapsto \mathit{location}(e)] & n = [\![\mathsf{final}\ v = e]\!] \ \wedge\ \mathit{varBatch}(n)\ \not= v_b \\ \mathsf{nil} & \mathit{otherwise} \end{cases}
\end{aligned}
$$

Figure 3.3: Analysis of Java to identify local and remote variables

statement node in the control flow graph. For simplicity, we will assume in this paper that all assignments are statements; however, in Java they are actually expressions. The data flow analysis is defined in Figure 3.3.

The `batch` variable is remote. Variables only assigned outside the batch are static locals. Variables declared final and initialized with remote expressions are remote. All other variables inside a batch block are non-static locals. Assignments may change the mapping of a variable up the lattice of locations.

$$
\begin{aligned}
location([\![v]\!]) &= in[Stmt(v)](v) \\
location([\![e_1 \ \mathsf{op} \ e_2]\!]) &= location(e_1) \sqcup location(e_2) \\
location([\![o.m(\bar{e})]\!]) &= location(o) \\
location([\![\_]\!]) &= \begin{cases} \text{non-static local} & inBatch(\_) \\ \text{static local} & !\ inBatch(\_) \end{cases}
\end{aligned}
$$

Figure 3.4: Location of Java expressions

For this analysis, the only case where this happens is a variable mapped as a static local may be remapped as a non-static local. It cannot happen for variables mapped as remote, because final variables cannot be reassigned.

Figure 3.4 defines the *location* function which maps expressions to locations. To determine the location of a variable expression, the analysis looks up the variable name in the result of the data flow analysis flowing into the statement containing the variable expression. The mutual definition of *location* and *gen* introduces a cyclic dependency which is resolved by taking the fix point of the two functions starting with the bottom value of our location lattice (static locals). The location of a primitive operation is the join of the locations of the operands. The location of an instance method call expression is the location of the target of the method call. All other expressions inside or outside the batch statement are non-static local or static local respectively.

The *location* and data-flow functions depend on each other's values. This circular dependency is resolved using a fix-point algorithm. The location of all expressions and variables is initialized to be the least value in our lattice

(non-static local). Since the size of the location lattice is three, each location and data-flow function may be computed at maximum three times. Thus, the analysis is linear in the size of the control flow graph and the number of expressions.

3.2.2 Illegal Programs

The compiler rejects all programs in which the remote operations cannot be legally moved above the local operations. For example, parameter expressions in remote method calls cannot contain local variables defined within the batch. The compiler also rejects some programs in which moving the remote operations above the local operations might result in non-intuitive behavior. For example, parameter expressions in remote method calls should not have their value changed in the local operations. The following are considered illegal expressions by the compiler.

- Method invocations on remote values that have a parameter which is a non-static local expression or is not serializable.
- Expressions with remote locations inside of an `if` block where the condition is a local expression.
- Expressions with remote locations inside of a loop construct where the condition is local.
- Nested batch statements.

One design goal was to ensure that programmers could easily understand the semantics of the `batch` construct. To that end, our analysis uses a very simple local data flow analysis and is lexically scoped. This may allow non-intuitive programs to be accepted by the compiler, because they change the state of static local expressions via different threads, heap aliasing, or local method calls [67]. The following example shows a case where the compiler accepts a program that behaves non-intuitively from the point of view of the programmer.

```
StringBuilder sb = new StringBuilder();
sb.append("My Album");
batch(Music music : musicService) {
  m(sb);
  music.createAlbum("1", sb);
}
...
void m(StringBuilder sb) { sb.append(": Blues"); }
```

The programmer might expect that the remote method call `createAlbum` will be passed the string `"My Album: Blues"`, but in a remote batch it will be passed the string `"My Album"`, because the remote method call will occur first. Unfortunately Java reflection, virtual methods, and dynamic class loading all complicate whole program analysis. Our local lexical analysis trades off catching some non-intuitive behavior to gain simplicity, practicality, and locality.

3.3 Implementation

Remote Batch Invocation is implemented as a source to source translator which takes Java source code containing batch statements and translates it into vanilla Java source code. In the new Java program, batch blocks are partitioned into a remote block encoded as an AST object representing the language in Figure 3.2 and a local block expressed as Java code. In a previous version [83], the remote block was encoded into calls to BEST (implemented by Eli Tilevich and Yang Jiao) which extends the implementation of BRMI [146].

For RMI, the remote block AST is sent to the server using Java RMI and our server object executes and returns the results to the client. Other transport mechanisms are possible such as using a web services framework. Once the batch results are received, the client then executes the local code using the results as needed.

3.3.1 Partitioning

A RBI program is compiled using a source to source translator. The source to source translator is implemented as an extension to JastAddJ [61]. JastAddJ is a Java analysis framework based on JastAdd and written as a circular attribute grammar. JastAdd provides several useful features. As a circular attribute grammar, many static analyses can be expressed naturally and fixed point computations are handled by the JastAdd engine. In addition, JastAdd provides many aspect-oriented features which allow composition of different analyses and language features in a a modular fashion. The data flow

analysis is implemented on top of a control flow graph module written by the authors of JastAddJ for Java 1.4. We modified the their module slightly to add support for the new `batch` construct and to support Java 1.5. For each expression, the translator computes its location as described in Section 3.2.

The translator traverses the program abstract syntax tree (AST) downwards starting from the root AST node. Outside of a batch, the translator does not change the Java code. Inside a batch, the translator produces two code partitions, one for the remote operations and one for the local operations. The remote operations are represented as an AST object. Once the entire batch is translated, some boilerplate code to setup the batch is generated first, then the remote operations are inserted, then a call to execute the batch is generated, and finally the local operations are inserted. While translating code in a batch, the translator has two different modes of operation. Initially the translator is in local mode. Expressions in local mode produce no remote operations and produce themselves as local operations. Most statements behave similarly except for remote loops and remote conditionals which produce both remote and local operations. Once the translator reaches an expression whose location is remote, it binds that remote expression to a named *handle* and marks that remote expression as being required locally. The translator also adds a local operation which invokes the `get` method on the handle. In remote mode, the translator can safely assume all sub-expressions are remote operations.

Figure 3.5 shows a RBI program which uses many of the supported features. Figure 3.6 shows the translation into Java code. An interesting part

```
Service musicService = new Service("MusicCloud", Music.class);
batch(Music music : musicService) {
  final Album album = music.getAlbum("1");
  if (album.getTitle().startsWith("A")) {
    System.out.println("Tracks:");
    for (Track t : album.getTracks()) {
      System.out.print(' ');
      System.out.print(t.getTitle());
    }
  } else {
    System.out.print("Title does not start with A: "
      + album.getArtist().getName());
}}
```

Figure 3.5: RBI source code

of the translation is how conditionals and loops require both remote and local operations. The local analogue of a remote loop invokes the `Cursor.next` method which updates the value of the handles inside the loop.

3.3.2 Batch Execution

At runtime, the client constructs the AST object representing the remote operations and sends it to the server with the necessary inputs. Inputs must be Java serializable objects or Java primitive values so that they can be passed by copy. The server interprets the remote operations and stores the results in an map of handle name, value pairs. Only the handles which are required by the local code on the client will be sent back to the client. Operations inside of loops are handled specially. Instead of updating the value of a handle on each iteration, the value of a handle is added to a list which

```
Expression ast$ = new Sequence(new Expression[] { // Remote part
 new MethodInvocation("a$172055",false,new RootExpression(),
  new Expression[] {new Constant("1")},"getAlbum","Music"),
 new IfStmt(
  new MethodInvocation("v$3", true,
   new MethodInvocation("v$2",false,new Ref("a$172055","Album",false),
    new Expression[] {},"getTitle","Album"),
   new Expression[] {new Constant("A")},"startsWith","String"),
  new Sequence(new Expression[] { Loop("t$1$Cursor",
    MethodInvocation("v$4",false,Ref("a$172055","Album",false),
     new Expression[] {},"getTracks","Album"),
    Sequence(new Expression[] {
     new MethodInvocation("v$5",true,new Ref("t$1$Cursor","Track",false),
      new Expression[] {},"getTitle","Track")}))}),
    Sequence(new Expression[] { new MethodInvocation("v$7",true,
      MethodInvocation("v$6",false,new Ref("a$172055","Album",false),
       new Expression[] {},"getArtist","Album"),
      Expression[] {},"getName","Artist")}))});
BatchClient.BatchResults results$ = musicService.executeBatch(ast$);
Handle v$3 = results$.getHandles().get("v$3");;
CursorIterator t$1$Cursor = results$.getCursors().get("t$1$Cursor");
Handle v$5 = results$.getHandles().get("v$5");;
Handle v$7 = results$.getHandles().get("v$7");;

if((Boolean)v$0.get()) { // Local part
  System.out.println("Tracks:");
  while (t$1$Cursor.next()) {
    System.out.print(' '); System.out.print((String)v$1.get());
}} else {
  System.out.print("Title does not start with A: "
    + (String)v$2.get());
}}
```

Figure 3.6: Translation of Figure 3.5

represents the value of the handle on each iteration.

3.3.3 Result Interpretation

For each non-cursor client handle, the server returns a value, exception, or nothing. Values must be Java serializable objects or Java primitive values. The server returns no value for a client handle associated with an unexecuted remote operation such as one guarded by a false conditional statement. At most one handle is assigned an exception, because the the remote batch is terminated by the first exception. If a handle has an exception, rather than a value, then this exception is thrown when accessing its content. Each time `next` is called on a cursor, the Handle objects associated with it change their value to the next iteration value.

3.4 RBI-DB

So far, we have seen that RBI can batch a series of remote calls to a remote server. This can be applied directly to transparent persistence by placing an RBI server wrapper around the persistent database and batching calls off of a root persistent object. For relational databases, we can create a special root persistent object with getter methods for each table in the database. Each table getter method returns a set of objects representing the tuples in that table. The code listing in Figure 3.7 shows an example of such a root object. However, we are not done yet. Persistence and relational databases pose two additional challenges for batching.

```
interface Root {
  Set<Employee> getEmployees();
  Set<Department> getDepartments();
  Set<Company> getCompanies();
}
```

Figure 3.7: Example of root class synthesized for our data model.

- Remote loops over persistent data can have a large number of iterations. By default, RBI sends information about each iteration to the client. On the other hand, explicit query languages like SQL can *filter* data returned to the client.

- A remote batch consists of a simple functional program. A relational database cannot execute such a program efficiently; the query optimizer only works with SQL queries.

To address both these concerns, we developed RBI-DB which extends RBI to translate remote batches into SQL and execute that SQL instead of the remote batch. Since SQL queries are read-only, we will only consider getter methods on persistent classes to be remote. Setter method invocations will remain in the local batch block. For example, the batch block in Figure 3.8 expresses a simple SQL query.

This query iterates over all the employees in the database and performs the local giveBonus action for employees with a salary above 100000. RBI-DB partitions the batch block into a SQL query and local client code as show in Figure 3.9.

```
batch db (Root r : databaseService) {
  for (Employee e : r.getEmployees()) {
    if (e.getSalary() < 100000) {
      giveBonus(e.getName(), e.getSalary());
    }
  }
}
```

Figure 3.8: Code written for RBI-DB.

```
Criteria c = databaseService.getSession().createCriteria(
  Employee.class, "emp");
c.add(Restrictions.lt("salary", 100000));
c.setProjection(Projections.projectionsList()
  .add(Projections.property("name"))
  .add(Projections.property("salary")));
List results = c.list();
Handle name = new Handle();
name.setValues(getAliasColumn(results, "name"));
Handle salary = new Handle();
salary.setValues(getAliasColumn(results, "salary"));
Cursor c = new Cursor(results.size());
c.addHandle(name); c.addHandle(salary);

// Local
while (c.hasNext()) {
  if (salary.getValue() < 100000) {
    giveBonus(name.getValue(), salary.getValue());
  }
}
```

Figure 3.9: RBI-DB code partitioned into SQL and local code.

$$
\begin{aligned}
\texttt{QueryLoop} &\leftarrow \texttt{for}\ (v\ \in\ \texttt{Path}.\mathit{getCollAssoc}())\ \texttt{QueryLoopBodyStmt}* \\
\texttt{QueryLoopBodyStmt} &\leftarrow \texttt{QueryLoopFilter} \mid \texttt{Path} \mid \texttt{QueryLoop} \\
\texttt{Path} &\leftarrow v \mid \texttt{Path}.\mathit{getSingleAssoc}() \\
\texttt{QueryLoopFilter} &\leftarrow \texttt{if Cond QueryLoopBodyStmt}* \texttt{ ;} \\
\texttt{Cond} &\leftarrow \texttt{Path} \mid \texttt{Cond}\ \mathit{binop}\ \texttt{Cond} \mid \mathit{unop}\ \texttt{Cond} \mid \mathit{constant}
\end{aligned}
$$

Figure 3.10: Definition of Query Loops

The filter in Figure 3.8 on line 3 remains even though it is redundant. There are some cases where the condition is needed and so the current RBI-DB implementation would need an additional analysis to determine whether filter conditions can be safely removed.

3.4.1 Translating Remote Code to SQL

Our remote batch DSL can express many programs which cannot be converted into a single SQL statement. Of those programs that can be converted into SQL queries, our translator can recognizes a subset. QueryII [88] implements a similar idea by translating Java bytecode to SQL queries for particular patterns of Java code. The primary pattern our translator looks for is a query loop, which has structure show in Figure 3.10.

The collection used in the query loop must be a collection association getter invocation on a *path*. A path is single persistent value within a context and can either be a variable reference to loop variable or single association getter invocation on a path. The query loop body may contain paths, nested

query loops, and filters. A filter is an `if` statement with an empty else branch. The condition in the filter must be a path or an expression involving paths, unary operators, binary operators, static local variables, and constants. Static local variables are translated into parameters in the SQL query.

Given a query loop, it can be translated into SQL. A persistent collection access from the `Root` class is translated to top-level table in the SQL query from clause. A persistent collection access from a persistent entity's collection association translates to a left outer join in the SQL query. A filter is translated into a where clause in the SQL query. Paths accessed inside the loop correspond to SQL projections.

3.4.2 Future Work

One weakness of our query translation is that we do not support several important features of SQL such as aggregations, group by, order by, and subqueries. Embedded query languages such as LINQ [50] support these features explicitly. Some of these constructs such as group by could be inferred from a particular code pattern; however, aggregations and order by might need additional syntax or semantics to be added to Java.

3.5 Related Work

RBI combines two different techniques, program partitioning and batching. There has been extensive research into each of these techniques individually and we first focus on existing work in optimizing distributed systems

using batching.

3.5.1 RPC Critique

Even though Remote Procedure Call (RPC) [141] has been one of the most prevalent communication abstracts for building distributed systems, its shortcomings and limitations have been continuously criticized [139, 156, 124]. Recently some experts even express the sentiment that RPC has had an overwhelmingly harmful influence on distributed systems development and wish that a different communication abstraction had become dominant instead [152]. A frequently mentioned alternative for RPC is asynchronous messaging and events, including publish-subscribe abstractions [53].

Despite all the criticisms of RPC and its object-oriented counterparts, exposing distributed functionality through a familiar procedure call paradigm has unquestionable convenience advantages. Remote Batch Invocation is an attempt to address some of the limitations of RPC, while retaining its advantages, without introducing the complications of asynchronous processing imposed by message- and event-based abstractions.

Among the main criticisms of RPC is its attempt to eliminate the distinction between the local and remote computing models, with respect to latency, memory access, concurrency, and partial failure [156]. By combining multiple operations into a single batch, RBI reduces latency. By executing all remote operations on the server in bulk, RBI maintains the local memory access model for method parameters. As future work, a transactional execu-

tion model can be combined with RBI to achieve an all-or-nothing execution property. And while batch invocations in RBI are synchronous, the resulting execution model is explicit, giving the programmer a clear execution and performance model.

3.5.2 Mobile Code

Mobile object systems such as Emerald [23] reduce latency by moving active objects, rather than making multiple remote calls. JavaParty [119] migrates objects to adapt the distribution layout of an application to enhance locality. Ambassadors is a communication technique that uses object mobility [58] to minimize the aggregate latency of multiple inter-dependent remote methods. DJ [1] adds explicit programming constructs for direct type-safe code distribution, improving both performance and safety.

Mobile objects generally require sophisticated runtime support not only for moving objects and classes between different sites, but also for dealing with security issues. In RBI, Clients only gain access to interfaces that are reachable from the service root.

RBI uses mobile code, specifically remote evaluation [131], to execute remote code in one round-trip. The advantage of RBI with respect to mobile code is the preceding partitioning step. This can be understood by considering a translation from RBI to mobile code. A `batch` statement could be implemented using mobile code by writing two mobile classes, one that is sent from the client to the server to execute the remote operations, and another that is

sent from the server back to the client to transport the results in bulk to the client. The first class would contain member variables to store all the local data sent to the server, and a method body to execute on the server. At the start of this method an instance of the second class is created and populated with data created by the remote method. At the end of the method the result object is sent back to the client. A custom pair of classes is needed for each `batch` statement in the program. While mobile code is more flexible and powerful than RBI, it can also be more work to use this power to implement common communication patterns.

3.5.3 Implicit Batching

Batched futures [26] in Thor reduce the aggregate latency of multiple remote methods. If remote methods are restructured to return futures, they can be batched. The invocation of the batch can be delayed until a value of any of the batched futures is used in an operation that needs its value. There are several different client invocation patterns that cannot be batched in this model. For example, unrelated remote method calls will not be batched together.

Future RMI [4] communicates asynchronously to speed up RMI in Grid environments, when one remote method is invoked on the result of another. Remote results of a batch are not transferred over the network, remaining on the server to be used for subsequent method invocations.

Yeung and Kelly [43] use byte-code transformations to delay remote

methods calls and create batches at runtime. A static analysis determines when batches must be flushed.

In all of these implicit batching techniques, it is not clear how to support loops, branches, and exceptions as in Remote Batch Invocation. In addition, small changes in the program, for example introducing an assignment to a local variable, or an exception handler, can cause a batch to be flushed. This means the performance is very sensitive to the ordering of remote and local operations. On the other hand, Remote Batch Invocation automatically tries to reorder remote and local operations to maintain a single batch, while checking that the reordering makes sense.

3.5.4 Explicit Batching

Software design patterns [66] for *Remote Façade* and *Data Transfer Object* (also called Value Objects [5]) can be used to optimize remote communication. A *Remote Façade* allows a service to support specific client call patterns using a single remote invocation. Different Remote Façades may be needed for different clients. Remote Batch Invocation provides a custom Remote Façade for each client as long as the client call pattern is supported as a single batch. A *Data Transfer Object* is a `Serializable` class that provides block transfer of data between client and server. As with the Remote Façade, different kinds of Data Transfer Objects may be needed by different clients. Remote Batch Invocation constructs an appropriate value object on the fly, automatically, as needed by a particular situation. Remote Batch Invocation

also generalizes the concept of a data transfer object to support transfer of data from arbitrary collections of objects.

The DRMI system [105] aggregates RMI calls as a middleware library much like BEST. DRMI uses special interfaces to record and delay the invocation of remote calls. DRMI only supports simple call aggregation and simple branching, while Remote Batch Invocation and BEST also support cursors, primitive operations, and exception handling. Like BEST, DRMI requires that the programmer partition the remote and local operations themselves. This often forces the programmer to replicate loops and conditionals manually, whereas Remote Batch Invocation offers a more flexible style of programming and relies on the source to source translator to partition the program into remote and local operations.

Detmold and Oudshoorn [59] present analytic performance models for RPC and its optimizations including batched futures as well as a new optimization construct termed *a responsibility*. Their analytic models could be extended to model the performance properties of the new optimization constructs of Remote Batch Invocation such as cursors and branching.

Sometimes a communication protocol defines batches directly, as is in the compound procedure in Network File System (NFS) version 4 Protocol [142], which combines multiple NFS operations into a single RPC request. The compound procedure in NFS is not a general-purpose mechanism; the calls are independent of each other, except for a hard-coded current filehandle that can be set and used by operations in the batch. There is also a single built-

in exception policy. Web Services are often based on transfer of documents, which can be viewed as batches of remote calls [154, 48].

Cook and Barfield [48] showed how a set of hand-written wrappers can provide a mapping between object interfaces and batched calls expressed as a web service document. Remote Batch Invocation automates the process of creating the wrappers and generalizes the technique to support branching, cursors, and exception handling. As a result, Remote Batch Invocation scales as well as an optimized web service, while providing the raw performance benefits of RPC [57]. Web services choreography [118] defines how Web services interact with each other at the message level. Remote Batch Invocation can be seen as a choreography facility for distributed objects.

Zondervan [163] expanded the batched futures in Thor to allow for control flow and value dependencies using *promises*. Promises are explicitly used by the client to wrap values including primitives and delay their computation. Special conditional and loop structures implemented using C++ macros allow the program to execute these statements remotely. Assignment is not supported directly, instead assignment can be simulated using a persistent *cell*. A cell is holder object with `get` and `set` methods. Assignment can be supported similarly in RBI. The client can explicitly force a promise to be executed, which forces all dependent computations to be sent to the Thor persistent store. Tilevich et al. developed BRMI [146] which implements much of the same ideas in Java using code generation and a more programmatic approach to control flow statements. Bao et al. [16] implemented batching for the

business process execution language (BPEL) using a static analysis.

3.5.5 Automatic Partitioning

Remote Batch Invocation can be seen as a language level abstraction for automatic application partitioning. Although RBI uses the partitioning for batching, the idea of partitioning is useful in many areas.

One line of research has explored coarse grained program partitioning. The programmer, by means of a GUI or a configuration file, designates different parts of a centralized application, typically at a class or object granularity, to run on different network nodes. The resulting distribution specification then parameterizes a compiler-based tool that automatically rewrites the centralized application for distributed execution. To introduce distribution, a partitioning tool may need to both change the structure of the application (e.g., to introduce a proxy indirection) and add middleware functionality (e.g., to replace local calls with remote ones). In the Java world, recent automatic partitioning tools include Addistant [140], Pangaea [130], and one of the co-author's J-Orchestra [144]. Addistant and J-Orchestra partition programs at a class granularity; Pangaea can partition at the individual object level. J-Orchestra addresses the challenges of partitioning programs safely in the presence of unmodifiable code that comes as part of their runtime systems.

Automatic program partitioning has also been applied at finer granularaties. Swift [44] partitions Java programs into a web application backend and Javascript at the Java statement level. Constraints on the locations of

statements is inferred from information flow policies and the placement of statements is optimized to minimize round-trips with respect to those constraints. Similarly, RBI infers the location of statements and expressions from a forward data-flow analysis.

3.5.6 Partitioning for Persistence

Kleisli [161] [160] is a language for writing distributed queries. A distributed query is expressed in the nested relational calculus (NRC) and can use disparate data sources such as hierarchical databases, relational databases, and in-memory data structures. Kleisli optimizes the distributed query through a series of rewrite rules. One important optimization is that Kleisli partitions a query into sub-queries in the native language of the data sources to take advantage of their query optimizers and reduce the data processed locally. For relational databases, Kleisli tries to extract sub-queries which are expressible in SQL; it supports projections, conditions, and joins. NRC upon which Kleisli is built is not a object-oriented nor turing complete so partitioning is less complex. For example, because NRC is stateless, the data flow analysis is simplified. Unlike RBI-DB, Kleisli does not guarantee a certain number of queries will be executed, although the programmer may examine the optimized query to predict its behavior. Kleisli has been successfully used in the field of bioinformatics.

Query Extraction [158] is a system for extracting database queries from Java code that traverses persistent object structures. Query Extraction per-

forms a similar analysis to RBI-DB to extract the code operating over persistent data and converts that code's loops and conditions to *join* and *where* clauses in database queries. RBI-DB allows the programmer to control where and when queries are executed. On the other hand, Query Extraction uses an implicit interprocedural analysis and so is able to compose queries across methods. Query Extraction is also an implicit batching mechanism. The programmer does not include batch blocks in their programs. Instead, the analysis automatically detects persistent access and places a query at the beginning of the enclosing method. More complicated query placement strategies are possible, but have not been explored. Both approaches have their merits. RBI-DB is simpler to implement and understand, but only creates queries intraprocedurally and requires the programmer to place queries explicitly using batch blocks.

Both Query Extraction and RBI-DB do not support grouping, aggregation, and ordering in the SQL queries extracted. We believe it might be easier for RBI-DB to support these features, because it is already committed to changing the Java language.

3.5.7 Asynchronous Remote Invocation

Another approach to optimizing distributed communication is dispatching remote calls asynchronously. One example is ProActive [14]. An asynchronous remote call in ProActive returns a future; a placeholder for to be computed results. When a client tries to resolve the future's actual value, the

client blocks until the result is available.

Although asynchronous remote invocations can optimize many patterns in client-server communication, they offer no performance improvements for chains of remote calls (i.e., `o.m1().m2()`). Compared to asynchronous invocation, the RBI programming model does not involve futures and can combine chains of remote calls into a batch, thus improving their performance.

Although the current version of RBI does not take advantage of concurrent processing, in the future we could have the server execute method calls asynchronously when possible. In addition, the client can send the entire batch asynchronously. Thus asynchronicity is orthogonal to batching.

3.6 Conclusion

Batching in and of itself is not new. Promises, BRMI, and many forms of mobile code support the remote batches used in RBI. Instead the novel contribution in RBI is combining program partitioning with batching. The benefits of RBI include:

- RBI provides a strong performance model. One server round-trip is executed for each lexical batch block.
- RBI allows multiple remote operations to be combined in a *batch* which is executed in a single round-trip to a remote server. A batch supports both control and data flow dependencies between remote operations.

As a consequence, the remote server may provide a flexible fine-grained interface.

- RBI allows the programmer to mix remote and local operations naturally. The compiler separates the remote operations and takes care of transferring multiple inputs to the remote server and interpreting the multiple outputs.

RBI was implemented as a Java extension using a source to source translator. RBI combines the convenience and flexibility of fine-grained interfaces with the performance advantages of coarser-grained interfaces. In addition, the RBI stateless execution model aligns well with the increasingly prevalent service-oriented architectures, a rapidly-emerging industry standard.

We also extended RBI to address transparent persistent with RBI-DB. RBI-DB extends RBI to compile the remote batch into SQL instead of sending it to custom RBI server. This allows RBI-DB to take advantage of a relational database's query optimizer for efficient evaluation of queries.

Chapter 4

Conclusion

Transparent persistence allows the programmer to write persistent programs in an elegant and modular fashion. However, naive execution of transparent persistence results in many calls to the database which results in poor performance if the database is in a separate process or on a separate machine. In addition, transparent persistence does not allow the program to take advantage of a database's query optimizations. Call-level interfaces allow the program to directly issue queries, but are clumsy to use and hard to modularize.

In this thesis we have tried to make transparent persistence more practical especially using relational databases. Relational databases are pervasive and providing a clean programming model for accessing them would be very useful in enterprise applications. We have examined two approaches.

AutoFetch attempts to improve existing tools which offer a hybrid of transparent persistence and call-level interfaces. Because it uses a dynamic analysis and a plugin architecture, it is easy for programmers to incorporate it into their programs. Already, we have seen other programmers implement and use the ideas of AutoFetch.

RBI addresses the general problem of batching remote calls in a distributed system and RBI-DB specializes RBI for handling transparent persistence. Instead of relying on libraries and code-generation, RBI changes the Java language by adding the batch construct. This allows the programmer to inform the compiler what performance characteristics they expect from the program's execution. A partitioning algorithm allows the programmer to mix local and remote code. Otherwise, the programmer must manually separate local and remote code, write custom data structures to transfer data, and replicate control flow.

Both tools help programmers express more of their programs in an object-oriented style while taking advantage of the power of relational databases.

Bibliography

[1] Alexander Ahern and Nobuko Yoshida. Formalising Java RMI with explicit code mobility. In *Proc. of OOPSLA '05*, pages 403–422, New York, NY, USA, 2005. ACM.

[2] Jung-Ho Ahn and Hyoung-Joo Kim. Seof: An adaptable object prefetch policy for object-oriented database systems. In *Proceedings of the 13th International Conference on Data Engineering*, pages 4–13, 1997.

[3] M. Alt and S. Gorlatch. Future-based RMI: Optimizing compositions of remote method calls on the grid. *Proc. of the Euro-Par*, 2790:427–430, 2003.

[4] M. Alt and S. Gorlatch. Adapting Java RMI for grid computing. *Future Generation Computer Systems*, 21(5):699–707, 2005.

[5] D. Alur, J. Crupi, and D. Malks. *Core J2EE Patterns: Best Practices and Design Strategies*. Prentice Hall PTR, 2003.

[6] Scott Ambler. Mapping objects to relational databases. web article, 1998.

[7] ANSI/INCITS. Database languages - SQLJ - part 1: SQL routines using the Java programming language. Technical Report 331.1-1999, ANSI/INCITS, 1999.

[8] M. P. Atkinson, M. J. Jordan, L. Daynè, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Proceedings of the Workshop on Persistent Object Systems (POS)*, pages 33–47, 1996.

[9] M. P. Atkinson and R. Morrison. Orthogonally persistent object systems. *VLDB Journal*, 4(3):319–401, 1995.

[10] Malcolm Atkinson, David Dewitt, David Maier, Francois Bancilhon, Klaus Dittrich, and Stanley Zdonik. *The object-oriented database system manifesto*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1994.

[11] Malcolm P. Atkinson. Persistence and java - a balancing act. In *Proceedings of the International Symposium on Objects and Databases*, pages 1–31, London, UK, 2001. Springer-Verlag.

[12] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Comput. Surv.*, 19(2):105–170, 1987.

[13] Malcolm P. Atkinson, Laurent Daynes, Mick J. Jordan, Tony Printezis, and Susan Spence. An orthogonally persistent Java. *SIGMOD Record*, 25(4):68–75, 1996.

[14] Laurent Baduel, Françoise Baude, Denis Caromel, Arnaud Contes, Fabrice Huet, Matthieu Morel, and Romain Quilici. *Grid Computing: Soft-*

ware Environments and Tools, chapter Programming, Deploying, Composing, for the Grid. Springer-Verlag, January 2006.

[15] Thomas Ball and James R. Larus. Efficient path profiling. In *International Symposium on Microarchitecture*, pages 46–57, 1996.

[16] Liang Bao, Ping Chen, Xiang Zhang, Sheng Chen, Shengming Hu, and Yang Yang. Batch invocation of web services in bpel process. In *ICSOC '08: Proceedings of the 6th International Conference on Service-Oriented Computing*, pages 511–516, Berlin, Heidelberg, 2008. Springer-Verlag.

[17] Douglas Barry and Torsten Stanienda. Solving the Java object storage problem. *Computer*, 31(11):33–40, 1998.

[18] Philip A. Bernstein, Shankar Pal, and David Shutt. Context-based prefetch for implementing objects on relations. In *The VLDB Journal*, pages 327–338, 1999.

[19] Gavin Bierman, Erik Meijer, and Wolfram Schulte. The essence of data access in cω. In *European Conference on Object-Oriented Programming*, 2005.

[20] K.P. Birman. Like it or not, Web services are distributed objects. *Communications of the ACM*, 47(12):60–62, 2004.

[21] A.D. Birrell and B.J. Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems (TOCS)*, 2(1):39–59, 1984.

[22] A. Black, N. Hutchinson, E. Jul, H. Levy, and L. Carter. Distribution and Abstract Types in Emerald. *IEEE Transactions on Software Engineering*, 13(1):65–76, 1987.

[23] Andrew P. Black, Norman C. Hutchinson, Eric Jul, and Henry M. Levy. The development of the Emerald programming language. In *HOPL III*, pages 11–1–11–51, 2007.

[24] Stephen Blackburn and John N. Zigman. Concurrency — the fly in the ointment? In *POS/PJW*, pages 250–258, 1998.

[25] Toby Bloom and Stanley B. Zdonik. Issues in the design of object-oriented database programming languages. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 441–451. ACM Press, 1987.

[26] P. Bogle and B. Liskov. Reducing cross domain call overhead using batched futures. *ACM SIGPLAN Notices*, 29(10):341–354, 1994.

[27] Michael D. Bond and Kathryn S. McKinley. Probabilistic calling context. In *OOPSLA '07: Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, pages 97–112, New York, NY, USA, 2007. ACM.

[28] G. Booch. *Object-oriented analysis and design.* Addison-Wesley, 1996.

[29] Evgueni Brevnov, Yuri Dolgov, Boris Kuznetsov, Dmitry Yershov, Vyacheslav Shakin, Dong-Yuan Chen, Vijay Menon, and Suresh Srinivas.

Practical experiences with java software transactional memory. In *PPoPP '08: Proceedings of the 13th ACM SIGPLAN Symposium on Principles and practice of parallel programming*, pages 287–288, New York, NY, USA, 2008. ACM.

[30] N. Brown and C. Kindel. Distributed Component Object Model Protocol–DCOM/1.0, 1998. Redmond, WA, 1996.

[31] Brendon Cahoon and Kathryn S. McKinley. Data flow analysis for software prefetching linked data structures in java. In *In International Conference on Parallel Architectures and Compilation Techniques*, pages 280–291, 2001.

[32] Michael J. Carey and David J. DeWitt. Of objects and databases: A decade of turmoil. In *Proceedings of the 22th International Conference on Very Large Data Bases*, pages 3–14. Morgan Kaufmann Publishers Inc., 1996.

[33] Michael J. Carey, David J. DeWitt, Chander Kant, and Jeffrey F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Proc. of ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications*, pages 414–426. ACM Press, 1994.

[34] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The 007 benchmark. *SIGMOD Rec.*, 22(2):12–21, 1993.

[35] Michael J. Carey, David J. DeWitt, and Jeffrey F. Naughton. The OO7 benchmark. In Peter Buneman and Sushil Jajodia, editors, *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 12–21. ACM Press, 1993.

[36] Michael J. Carey, David J. DeWitt, Jeffrey F. Naughton, Mohammad Asgarian, Paul Brown, Johannes E. Gehrke, and Dhaval N. Shah. The BUCKY object-relational benchmark. In *Proceedings of the 1997 ACM SIGMOD international conference on Management of data*, pages 135–146. ACM Press, 1997.

[37] R. G. G. Cattell, Douglas K. Barry, Mark Berler, Jeff Eastman, David Jordan, Craig Russell, Olaf Schadow, Torsten Stanienda, and Fernando Velez, editors. *The Object Data Standard ODMG 3.0*. Morgan Kaufmann, January 2000.

[38] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of EJB applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261. ACM Press, 2002.

[39] Emmanuel Cecchet, Julie Marguerite, and Willy Zwaenepoel. Performance and scalability of ejb applications. In *Proceedings of the 17th ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 246–261. ACM Press, 2002.

[40] Davor Cengija. Hibernate your data. *onJava.com*, 2004.

[41] E. E. Chang and R. H. Katz. Exploiting inheritance and structure semantics for effective clustering and buffering in an object-oriented dbms. In *SIGMOD '89: Proceedings of the 1989 ACM SIGMOD international conference on Management of data*, pages 348–357. ACM Press, 1989.

[42] Zhiyuan Chen and Praveen Seshadri. An algebraic compression framework for query results. In *ICDE*, pages 177–188, 2000.

[43] K. Cheung Yeung and PK Kelly. Optimising Java RMI Programs by Communication Restructuring. In *ACM Middleware Conference*. Springer, 2003.

[44] Stephen Chong, Jed Liu, Andrew C. Myers, Xin Qi, K. Vikram, Lantian Zheng, and Xin Zheng. Secure web application via automatic partitioning. In *SOSP '07: Proceedings of twenty-first ACM SIGOPS symposium on Operating systems principles*, pages 31–44, New York, NY, USA, 2007. ACM.

[45] E. Christensen, F. Curbera, G. Meredith, and S. Weerawarana. *Web Services Description Language (WSDL) 1.1*, March, 2001.

[46] William Cook and Siddhartha Rai. Safe query objects: Statically typed objects as remotely executable queries. Technical Report TR-04-17, UT Austin Computer Science, May 2004.

[47] William R. Cook and Siddhartha Rai. Safe query objects: statically typed objects as remotely executable queries. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 97–106. ACM Press, 2005.

[48] W.R. Cook and J. Barfield. Web Services versus Distributed Objects: A Case Study of Performance and Interface Design. In *the IEEE International Conference on Web Services (ICWS'06)*, pages 419–426, 2006.

[49] George Copeland and David Maier. Making Smalltalk a database system. In *Proceedings of the 1984 ACM SIGMOD international conference on Management of data*, pages 316–325. ACM Press, 1984.

[50] Microsoft Corporation. The LINQ project. `msdn.microsoft.com/netframework/future/linq`.

[51] Kenneth M. Curewitz, P. Krishnan, and Jeffrey Scott Vitter. Practical prefetching via data compression. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data (SIGMOD '93)*, 1993.

[52] M. Dahm. Doorastha: a step towards distribution transparency. In *Proceedings of JIT*, 2000.

[53] C.H. Damm, P.T. Eugster, and R. Guerraoui. Linguistic support for distributed programming abstractions. In *Distributed Computing Sys-*

tems. Proceedings. 24th International Conference on, pages 244–251, 2004.

[54] J Darmont and M Schneider. Benchmarking oodbs with a generic tool. *Journal of Database Management*, 11(3):16–27, Jul-Sept 2000.

[55] D. Davis and MP Parashar. Latency Performance of SOAP Implementations. In *Cluster Computing and the Grid, 2002. 2nd IEEE/ACM International Symposium on*, pages 407–407, 2002.

[56] Mark Stuart Day. *Client Cache Management in a Distributed Object Database*. PhD thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1995.

[57] C. Demarey, G. Harbonnier, R. Rouvoy, and P. Merle. Benchmarking the Round-Trip Latency of Various Java-Based Middleware Platforms. *Studia Informatica Universalis Regular Issue*, 4(1):7–24, 2005.

[58] H. Detmold, M. Hollfelder, and M.J. Oudshoorn. Ambassadors: structured object mobility in worldwide distributed systems. In *Proc. of ICDCS'99*, pages 442–449, 1999.

[59] H. Detmold and M.J. Oudshoorn. Communication Constructs for High Performance Distributed Computing. In *Proceedings of the 19th Australasian Computer Science Conference*, pages 252–261, 1996.

[60] Jacques-Antoine Dub, Rick Sapir, and Peter Purich. Oracle Application Server TopLink application developers guide, 10g (9.0.4). Oracle Corporation, 2003.

[61] Torbjörn Ekman and Görel Hedin. The JastAdd Extensible Java Compiler. *SIGPLAN Not.*, 42(10):1–18, 2007.

[62] R. Elfwing, U. Paulsson, and L. Lundberg. Performance of SOAP in Web Service environment compared to CORBA. In *Software Engineering Conference, 2002. Ninth Asia-Pacific*, pages 84–93, 2002.

[63] Thomas Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall, Upper Saddle River, NJ, USA, 2005.

[64] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext Transfer Protocol–HTTP/1.1, 1999.

[65] Joseph A. Fisher and Stefan M. Freudenberger. Predicting conditional branch directions from previous runs of a program. In *ASPLOS-V: Proceedings of the fifth international conference on Architectural support for programming languages and operating systems*, pages 85–95. ACM Press, 1992.

[66] Martin Fowler. *Patterns of Enterprise Application Architecture*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.

[67] Richard Gabriel. Is worse really better? *Journal of Object-Oriented Programming (JOOP)*, 5(4):501–538, 1992.

[68] J.J. Garrett. Ajax: A New Approach to Web Applications. *Url: http://www. adaptivepath. com/publications/essays//archives/000385*, 22, 2005.

[69] A. Gokhale, B. Kumar, and A. Sahuguet. Reinventing the Wheel? CORBA vs. Web Services. In *WWW 2002, The Eleventh International World Wide Web Conference, Honolulu, Hawaii, USA*, pages 7–11, 2002.

[70] Google Inc. *Google Web Toolkit*, 2007.

[71] C. Gould, Z. Su, and P. Devanbu. Static checking of dynamically generated queries in database applications. In *Proceedings, 26th International Conference on Software Engineering (ICSE)*. IEEE Press, 2004.

[72] William Grosso. Learning command objects and RMI. *http://www.onjava.com/pub/a/onjava/2001/10/17/rmi.html*, 2001.

[73] David Alan Halls. *Applying Mobile Code to Distributed Systems*. PhD thesis, University of Cambridge, Cambridge, England, 1997.

[74] R.H. Halstead. MULTILISP: a language for concurrent symbolic computation. *ACM Transactions on Programming Languages and Systems*, 7(4):501–538, 1985.

[75] Graham Hamilton and Rick Cattell. JDBC™: A Java SQL API. Sun Microsystems, 1997.

[76] Wook-Shin Han, Yang-Sae Moon, and Kyu-Young Whang. PrefetchGuide: capturing navigational access patterns for prefetching in client/server object-oriented/object-relational DBMSs. *Information Sciences*, 152(1):47–61, 2003.

[77] Wook-Shin Han, Yang-Sae Moon, Kyu-Young Whang, and Il-Yeol Song. Prefetching based on type-level access pattern in object-relational DBMSs. In *Proceedings of the 17th International Conference on Data Engineering*, pages 651–660. IEEE Computer Society, 2001.

[78] P. Heller. *Java 1.1 developer's handbook*. SYBEX San Francisco, CA, 1997.

[79] Hibernate reference documentation. `http://www.hibernate.org/hib_docs/v3/reference/en/html`, May 2005.

[80] Uwe Hohenstein. Bridging the Gap Between C++ and Relational Databases. In *European Conference on Object-Oriented Programming*, 1996.

[81] Uwe Hohenstein, Volkmar Plesser, and Rainer Heller. Evaluating the performance of object-oriented database systems by means of a concrete application. *Theor. Pract. Object Syst.*, 5(4):249–261, 1999.

[82] Antony L. Hosking and J. Eliot B. Moss. Towards Compile-Time Optimisations for Persistence. In Alan Dearle, Gail M. Shaw, and Stanley B.

Zdonik, editors, *Proceedings of the International Workshop on Persistent Object Systems*, pages 17–27, Martha's Vineyard, Massachusetts, September 1990. Morgan Kaufmann.

[83] Ali Ibrahim, Yang Jiao, William R. Cook, and Eli Tilevich. Remote batch invocation for compositional object services. In *European Conference on Object-Oriented Programming*, 2009. to appear.

[84] Ali H. Ibrahim and William R. Cook. Automatic prefetching by traversal profiling in object persistence architectures. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 50–73. Springer Berlin / Heidelberg, 2006.

[85] Tatsushi Inagaki, Tamiya Onodera, Hideaki Komatsu, and Toshio Nakatani. Stride prefetching by dynamically inspecting objects. In *PLDI '03: Proceedings of the ACM SIGPLAN 2003 conference on Programming language design and implementation*, pages 269–277, New York, NY, USA, 2003. ACM.

[86] INCITS/ISO/IEC. Information technology - database languages - SQL - part 5: Host language bindings (SQL/Bindings). Technical Report 9075-5-1999, INCITS/ISO/IEC, 1999.

[87] ISO/IEC. Information technology - database languages - SQL - part 3: Call-level interface (SQL/CLI). Technical Report 9075-3:2003, ISO/IEC, 2003.

[88] Ming-Yee Iu and Willy Zwaenepoel. Queryll: Java database queries through bytecode rewriting. In *Middleware '06: Proceedings of the ACM/IFIP/USENIX 2006 International Conference on Middleware*, pages 201–218, New York, NY, USA, 2006. Springer-Verlag New York, Inc.

[89] Java Center. J2EE^{TM} patterns. Sun Microsystems, 2003.

[90] Mick Jordan. Comparative study of persistence mechanisms for the java platform. Technical Report TR-2004-136, Sun Microsystems, September 2004.

[91] M.B. Juric, B. Kezmah, M. Hericko, I. Rozman, and I. Vezocnik. Java RMI, RMI tunneling and Web services comparison and performance analysis. *ACM SIGPLAN Notices*, 39(5):58–65, 2004.

[92] Tom Keller, Goetz Graefe, and David Maier. Efficient assembly for complex objects. In *SIGMOD '91: Proceedings of the 1991 ACM SIGMOD international conference on Management of data*, pages 148–157, New York, NY, USA, 1991. ACM.

[93] Nils. Knafla. Analysing object relationships to predict page access for prefetching. In *Eighth International Workshop on Persistent Object Systems: Design, Implementation and Use, POS-8*, 1998.

[94] C. Koelbel and P. Mehrotra. Compiling global name-space parallel loops for distributed execution. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):440–451, 1991.

[95] Dirk Krafzig, Karl Banke, and Dirk Slama. *Enterprise SOA : service-oriented architecture best practices*. Prentice Hall, 2005.

[96] Charles Lamb, Gordon Landis, Jack A. Orenstein, and Daniel Weinreb. The ObjectStore database system. *Commun. ACM*, 34(10):50–63, 1991.

[97] Karl J. Lieberherr, Boaz Patt-Shamir, and Doug Orleans. Traversals of object structures: Specification and efficient implementation. *ACM Trans. Program. Lang. Syst.*, 26(2):370–412, 2004.

[98] Mikko H. Lipasti, William J. Schmidt, Steven R. Kunkel, and Robert R. Roediger. SPAID: Software prefetching in pointer- and call-intensive environments. In *Proc. of the 28th Annual ACM/IEEE International Symposium on Microarchitecture*, pages 231–236, 1995.

[99] B. Liskov, A. Adya, M. Castro, S. Ghemawat, R. Gruber, U. Maheshwari, A. C. Myers, M. Day, and L. Shrira. Safe and efficient sharing of persistent objects in Thor. In *Proceedings of the 1996 ACM SIGMOD international conference on Management of data*, pages 318–329. ACM Press, 1996.

[100] B. Liskov and L. Shrira. Promises: linguistic support for efficient asynchronous procedure calls in distributed systems. In *PLDI 1988: Proceedings of the ACM SIGPLAN 1988 Conference on Programming Language Design and Implementation*, pages 260–267, New York, NY, USA, 1988. ACM Press.

[101] Barbara Liskov, Miguel Castro, Liuba Shrira, and Atul Adya. Providing persistent objects in distributed systems. *Lecture Notes in Computer Science*, 1628:230–??, 1999.

[102] Chi-Keung Luk and Todd C. Mowry. Compiler-based prefetching for recursive data structures. In *Architectural Support for Programming Languages and Operating Systems*, pages 222–233, 1996.

[103] David Maier. Representing database programs as objects. In François Bancilhon and Peter Buneman, editors, *Advances in Database Programming Languages, Papers from DBPL-1*, pages 377–386. ACM Press / Addison-Wesley, 1987.

[104] Jaydeep Marathe and Frank Mueller. Pfetch: software prefetching exploiting temporal predictability of memory access streams. In *MEDEA '08: Proceedings of the 9th workshop on MEmory performance*, pages 1–8, New York, NY, USA, 2008. ACM.

[105] Eduardo Marques. A study on the optimisation of Java RMI programs. Master's thesis, Imperial College of Science Technology and Medicine, University of London, 1998.

[106] Alonso Marquez, Stephen Blackburn, Gavin Mercer, and John N. Zigman. Implementing orthogonally persistent Java. In *Proceedings of the Workshop on Persistent Object Systems (POS)*, 2000.

[107] Bruce E. Martin. Uncovering database access optimizations in the middle tier with TORPEDO. In *Proceedings of the 21st International Conference on Data Engineering*, pages 916–926. IEEE Computer Society, 2005.

[108] V. Matena and M. Hapner. Enterprise Java Beans Specification 1.0. Sun Microsystems, 1998.

[109] F. Matthes, G. Schroder, and J.W. Schmidt. Tycoon: A scalable and interoperable persistent system environment. In M.P. Atkinson, editor, *Fully Integrated Data Environments*. Springer-Verlag, 1995.

[110] Erik Meijer and Wolfram Schulte. Programming with rectangles, triangles, and circles. In *Proceedings of Conference on XML*, 2003.

[111] Erik Meijer, Wolfram Schulte, and Gavin Bierman. Unifying tables, objects and documents. In *Proc. Declarative Programming in the Context of OO Languages, DP-COOL '03*, 2003.

[112] Microsoft Corporation. OLE DB/ADO: Making universal data access a reality, 1998.

[113] R Morrison, RCH Connor, GNC Kirby, DS Munro, MP Atkinson, QI Cutts, AL Brown, and A Dearle. The Napier88 persistent programming language and environment. In *Fully Integrated Data Environments*, pages 98–154. Springer, 1999.

[114] The Object Management Group (OMG). *The Common Object Request Broker: Architecture and Specification*, 1997.

[115] Mark Palmer and Stanley B. Zdonik. Fido: A cache that learns to fetch. In *Proceedings of the 17th International Conference on Very Large Data Bases*, 1991.

[116] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[117] David A. Patterson. Latency lags bandwith. *Commun. ACM*, 47(10):71–75, 2004.

[118] C. Peltz. Web services orchestration and choreography. *Computer*, 36(10):46–52, 2003.

[119] M. Philippsen and M. Zenger. JavaParty– transparent remote objects in Java. *Concurrency Practice and Experience*, 9(11):1225–1242, 1997.

[120] E. Pitt and K. McNiff. *Java.RMI: The Remote Method Invocation Guide*. Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 2001.

[121] Awais Rashid and Ruzanna Chitchyan. Persistence as an aspect. In *AOSD '03: Proceedings of the 2nd international conference on Aspect-oriented software development*, pages 120–129, New York, NY, USA, 2003. ACM Press.

[122] Roger Riggs, Jim Waldo, Ann Wollrath, and Krishna Bharat. Pickling State in the Java System. *Computing Systems*, 9(4):291–312, 1996.

[123] C. Russell. Java Data Objects (JDO) Specification JSR-12. Sun Microsystems, 2003.

[124] U. Saif and DJ Greaves. Communication primitives for ubiquitous systems or RPC consideredharmful. In *Distributed Computing Systems Workshop, 2001 International Conference on*, pages 240–245, 2001.

[125] J. W. Schmidt and F. Matthes. The rationale behind DBPL. In *Proc. of the 3rd Symposium on Mathematical Fundamentals of Database and Knowledge Base Systems (MFDBS-91)*, pages 389–395, Rostock, Germany, 1991.

[126] Joachim W. Schmidt and Florian Matthes. The DBPL project: advances in modular database programming. *Inf. Syst.*, 19(2):121–140, 1994.

[127] J.W. Schmidt, F. Matthes, and P. Valduriez. Building persistent application systems in fully integrated data environments: Modularization, abstraction and interoperability. In *Proceedings of Euro-Arch'93 Congress*. Springer Verlag, October 1993.

[128] M. Shapiro. Structure and Encapsulation in Distributed Systems: The Proxy Principle. In *Proceedings of ICDCS'86*, pages 198–204, 1986.

[129] V. Soloviev. An overview of three commercial object-oriented database management systems: ONTOS, ObjectStore, and O_2. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 21(1):93–105, 1992.

[130] A. Spiegel. *Automatic Distribution of Object Oriented Programs*. PhD thesis, FU Berlin, FB Mathematik und Informatik, 2002.

[131] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. Program. Lang. Syst.*, 12(4):537–564, 1990.

[132] Raible's wiki: StrutsResume. `http://raibledesigns.com/wiki/Wiki.jsp?page=StrutsResume`, March 2006.

[133] Muralidhar Subramanian and Vishu Krishnamurthy. Performance challenges in object-relational DBMSs. *IEEE Data Eng. Bull.*, 22(2):27–31, 1999.

[134] Sun Microsystems. *Java Core Reflection API and Specification*, 1997.

[135] Sun Microsystems. *Java Remote Method Invocation Specification*, 1997.

[136] Sun Microsystems. *Dynamic proxy classes specification*, 1999.

[137] Sun Microsystems. *Web Services Reference*, 2005.

[138] Sun Microsystems, February. *Java Object Serialization Specification*, 1997.

[139] Andrew Stuart Tanenbaum and Robbert van Renesse. A critique of the remote procedure call paradigm. In *EUTECO 88*, pages 775–783. North-Holland, 1988.

[140] M. Tatsubori, T. Sasaki, S. Chiba, and K. Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *European Conference on Object-Oriented Programming (ECOOP)*, 2001.

[141] B Tay and A Ananda. A survey of remote procedure calls. *Operating Systems Review*, 24(3):68–79, 1990.

[142] The Internet Society and Sun Microsystems. RFC 2339: An agreement between the Internet Society, the IETF, and Sun Microsystems, Inc. in the matter of NFS V.4 protocols, May 1998. Status: INFORMATIONAL.

[143] E. Tilevich and Y. Smaragdakis. NRMI: natural and efficient middleware. In *Proceedings of* 23^{rd} *International Conference on Distributed Computing Systems, 2003.*, pages 252–261, 2003.

[144] E. Tilevich and Y. Smaragdakis. J-Orchestra: Enhancing Java programs with distribution capabilities. *ACM Transactions on Software Engineering and Methodology*, 2009. in press.

[145] E. Tilevich and Y. Smaragdakis. NRMI: natural and efficient middleware. *IEEE Transactions on Parallel and Distributed Systems*, 19(2):x1–x14, Feb. 2008.

[146] Eli Tilevich, William R. Cook, and Yang Jiao. Explicit batching for distributed objects. In *International Conference on Distributed Computing Systems (ICDCS 2009)*, 2009. to appear.

[147] Eli Tilevich and Yannis Smaragdakis. Automatic application partitioning: The J-Orchestra approach. In *ECOOP 2002 Workshop on Mobile Object Systems*, 2002.

[148] Eli Tilevich and Yannis Smaragdakis. J-Orchestra: Automatic Java application partitioning. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, pages 178–204. Springer-Verlag, LNCS 2374, 2002.

[149] Ashutosh Tiwary, Vivek R. Narasayya, and Henry M. Levy. Evaluation of OO7 as a system and an application benchmark. In *OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, October 1995.

[150] Jeffrey D. Ullman, Hector Garcia-Molina, and Jennifer Widom. *Database Systems: The Complete Book*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001.

[151] Murali Venkatrao and Michael Pizzo. SQL/CLI – a new binding style for SQL. *SIGMOD Record*, 24(4):72–77, 1995.

[152] S. Vinoski. RPC Under Fire. *IEEE INTERNET COMPUTING*, pages 93–95, 2005.

[153] Guhan Viswanathan and James R. Larus. Compiler-directed shared-memory communication for iterative parallel applications. In *Supercomputing '96: Proceedings of the 1996 ACM/IEEE conference on Supercomputing (CDROM)*, page 10, Washington, DC, USA, 1996. IEEE Computer Society.

[154] W. Vogels. Web services are not distributed objects. *Internet Computing, IEEE*, 7(6):59–66, 2003.

[155] Philip Wadler. Deforestation: Transforming programs to eliminate trees. *Theoretical Computer Science*, 73:344–358, 1990.

[156] J. Waldo, A. Wollrath, G. Wyant, and S.C. Kendall. A Note on Distributed Computing. Technical report, Sun Microsystems, Inc. Mountain View, CA, USA, 1994.

[157] Noel Welsh, Francisco Solsona, and Ian Glover. SchemeUnit and SchemeQL: Two little languages. In *Third Workshop on Scheme and Functional Programming*, 2002.

[158] Ben Wiedermann, Ali Ibrahim, and William R. Cook. Interprocedural query extraction for transparent persistence. In *OOPSLA '08: Proceedings of the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, pages 19–36, New York, NY, USA, 2008. ACM.

[159] Darren Willis, David Pearce, and James Noble. Efficient object querying for java. In *Proceedings of the 20th European Conference on Object-Oriented Programming (ECOOP)*, pages 28–49. Springer Berlin / Heidelberg, 2006.

[160] Limsoon Wong. *Querying Nested Collections*. PhD thesis, University of Pennsylvania, 1994.

[161] Limsoon Wong. The functional guts of the kleisli query system. In *ICFP '00: Proceedings of the fifth ACM SIGPLAN international conference on Functional programming*, pages 1–10, New York, NY, USA, 2000. ACM.

[162] Daisuke Yokota, Shigeru Chiba, and Kozo Itano. A new optimization technique for the inspector-executor method. In *In Proc. of the International Conference on Parallel and Distributed Computing Systems (PDCS*, pages 706–711. ACTA Press, 2002.

[163] Quinton Zondervan. Increasing cross-domain call batching using promises and batched control structures. Master's thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, USA, 1994.

Vita

Ali Ibrahim was born in New York City, New York on Jun 5th 1980, the son of Dr. Hussein Ibrahim and Dr. Nihal Nounou. He received the Bachelor of Science and Master of Engineering degrees in Computer Science and Electrical Engineering from the Massachussetts Institute of Technology. He also received a Bachelor of Arts degree in Economics from the Massachussetts Institute of Technology. He worked for one year at Vecna Technologies before starting his doctoral studies at the University of Texas at Austin in the spring of 2004.

Permanent address: 3374 Lake Austin blvd. Apt D
Austin, Texas 78703

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.